

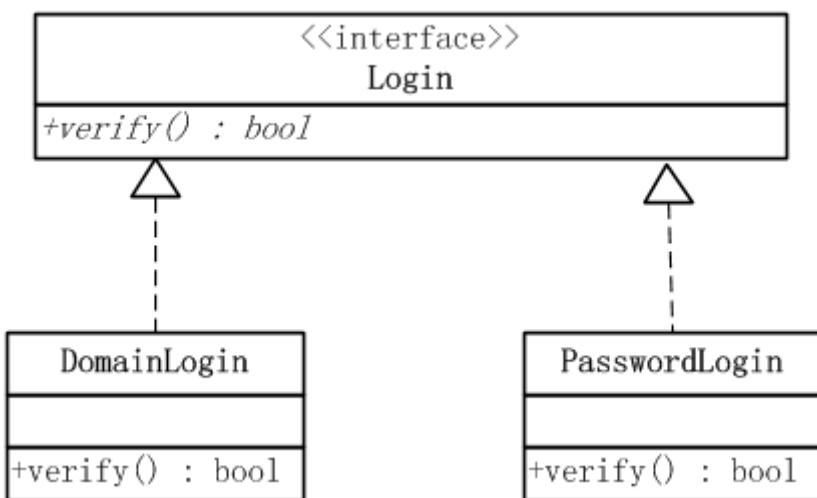
All in

— In Java

Java设计模式

1. 简单工厂模式

简单工厂模式是类的创建模式，又叫做静态工厂方法（Static Factory Method）模式。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。

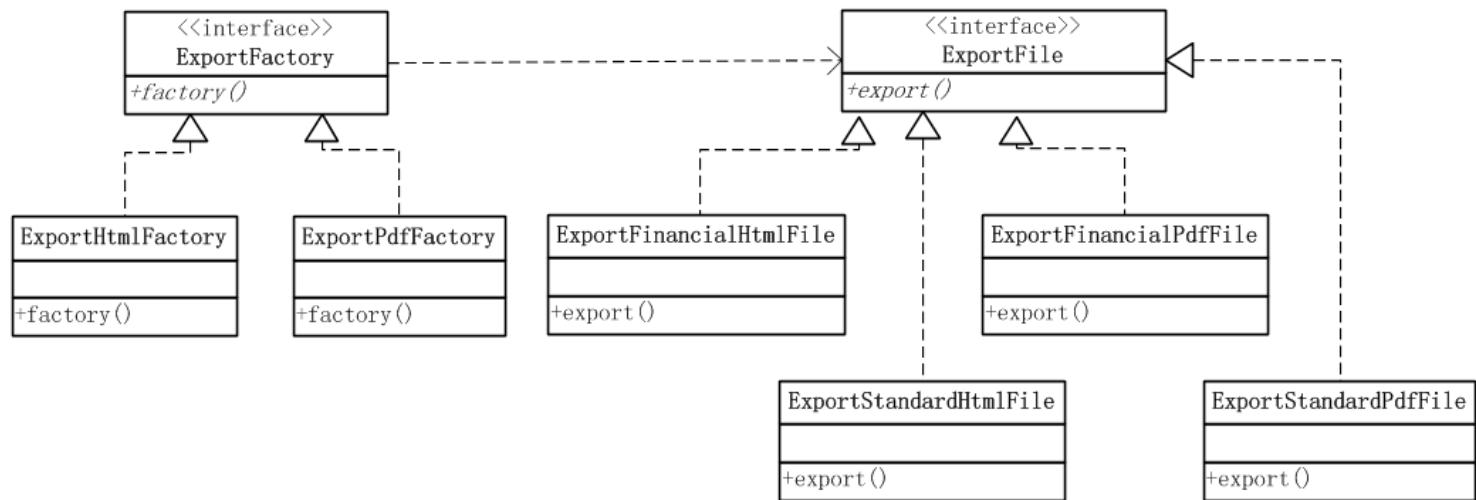


```
1. public class TestLogin {
2.     @Test
3.     public void test() {
4.         Login login = LoginSFM.factory(LoginSFM.TYPE.PASSWORD);
5.         assertTrue(login.getClass().equals(PasswordLogin.class));
6.     }
7. }
```

2. 工厂方法模式

工厂方法模式是类的创建模式，又叫做虚拟构造子(Virtual Constructor)模式或者多态性工厂 (Polymorphic Factory) 模式。

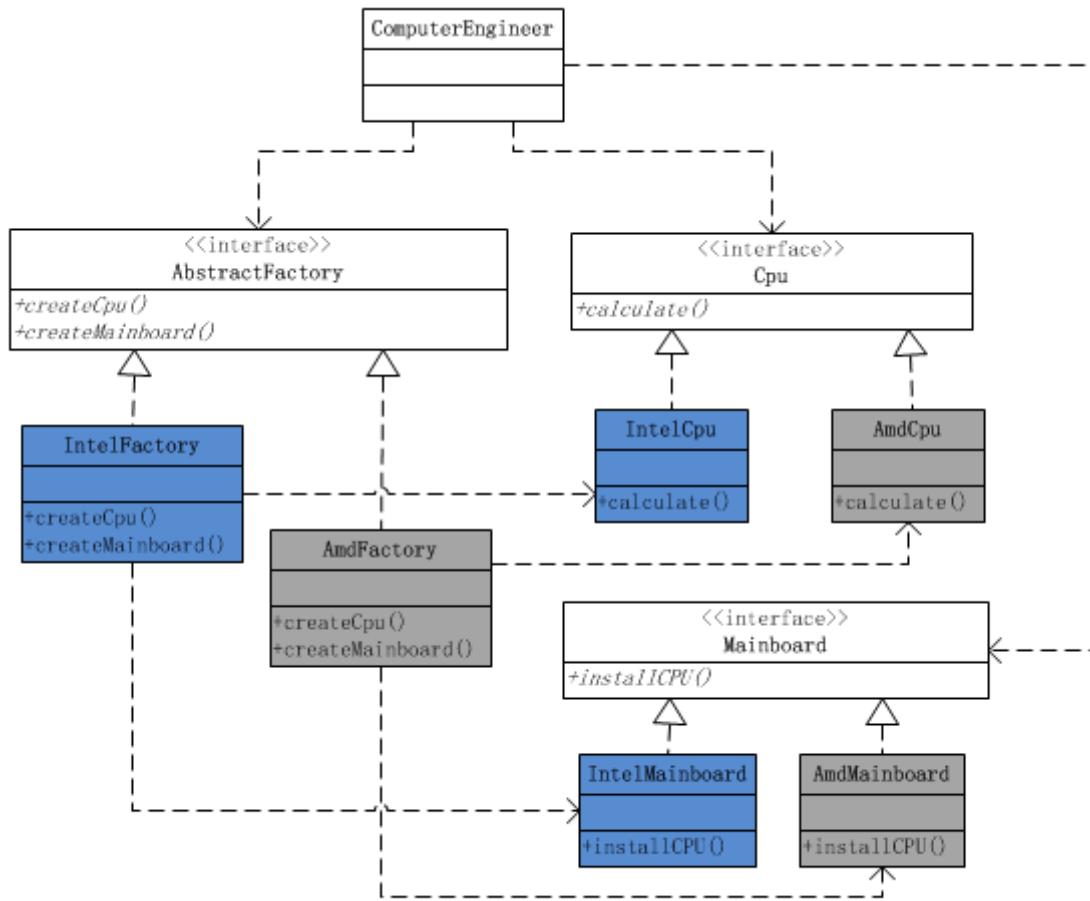
工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。



```
1. public class TestExport {
2.     @Test
3.     public void test() {
4.         String data = "";
5.         ExportFactory exportFactory = new ExportHtmlFactory();
6.         ExportFile ef = exportFactory.factory(Classify.Financial);
7.         ef.export(data);
8.     }
9. }
```

3. 抽象工厂模式

抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。



```
1. public class ComputerEngineer {
2.     /**
3.      * 定义组装机需要的CPU
4.      */
5.     private Cpu cpu = null;
6.     /**
7.      * 定义组装机需要的主板
8.      */
9.     private Mainboard mainboard = null;
10.    public void makeComputer(AbstractFactory af) {
11.        /**
12.         * 组装机器的基本步骤
13.         */
14.        // 1:首先准备好装机所需要的配件
15.        prepareHardwares(af);
16.        // 2:组装机器
17.        // 3:测试机器
18.        // 4: 交付客户
19.    }
20.    private void prepareHardwares(AbstractFactory af) {
21.        // 这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个
22.        // 可是，装机工程师并不知道如何去创建，怎么办呢？
23.        // 直接找相应的工厂获取
24.        this.cpu = af.createCpu();
25.        this.mainboard = af.createMainboard();
26.        // 测试配件是否好用
27.        this.cpu.calculate();
28.        this.mainboard.installCPU();
29.    }
30. }
```

```
1. public class Client {
```

```

2.     @Test
3.     public void test() {
4.         // 创建装机工程师对象
5.         ComputerEngineer cf = new ComputerEngineer();
6.         // 客户选择并创建需要使用的产品对象
7.         AbstractFactory af = new IntelFactory();
8.         // 告诉装机工程师自己选择的产品，让装机工程师组装电脑
9.         cf.makeComputer(af);
10.    }
11. }

```

4. 单例模式

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

单例模式的特点：

- 单例类只能有一个实例。
- 单例类必须自己创建自己的唯一实例。
- 单例类必须给所有其他对象提供这一实例。

不合理的两种写法：

```

1.  public class Singleton {
2.      private Singleton() {//确保单例不会在系统中的其他代码内被实例化
3.          System.out.println("Singleton is create");//创建单例的过程可能会比较慢
4.      }
5.      private static Singleton instance = new Singleton();
6.      public static Singleton getInstance() {
7.          return instance;
8.      }
9.  }

```

这种单例的实现方式非常简单，而且十分可靠。它唯一不足仅是无法对instance实例做延迟加载。加入单例的创建过程很慢，由于instance成员变量是static定义的，因此在JVM加载单例类时，单例对象就会被建立。

为了达到延迟加载，就需要引入延迟加载机制。

```

1.  public class LazySingleton {
2.      private LazySingleton() {
3.          System.out.println("LazySingleton is create");
4.      }
5.      private static LazySingleton instance = null;
6.      public synchronized static LazySingleton getInstance() {
7.          if(null == instance)
8.              instance = new LazySingleton();
9.          return instance;
10.     }
11. }

```

getInstance方法使用synchronized达到了同步，保证同时只有一个线程访问该方法，虽然实现了延迟加载的功能，但和第一种方法相比，它引入了同步关键字，因此在多线程环境中，它的时耗要远远大于第一种单例模式。因为第一种在运行前类加载时就已经创建了对象。

两种可行单例：

1. 常用方法

```
1. public class Singleton {
2.
3.     private Singleton(){} //确保实例不会在系统中的其他代码内被实例化
4.     /**
5.      *    类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例
6.      *    没有绑定关系，而且只有被调用到时才会装载，从而实现了延迟加载。
7.      */
8.     private static class SingletonHolder{
9.         /**
10.            * 静态初始化器，由JVM来保证线程安全
11.            */
12.            private static Singleton instance = new Singleton();
13.        }
14.
15.        public static Singleton getInstance(){
16.            return SingletonHolder.instance;
17.        }
18.    }
```

当getInstance方法第一次被调用的时候，它第一次读取SingletonHolder.instance，导致SingletonHolder类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建Singleton的实例，由于是静态的域，因此只会在虚拟机装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

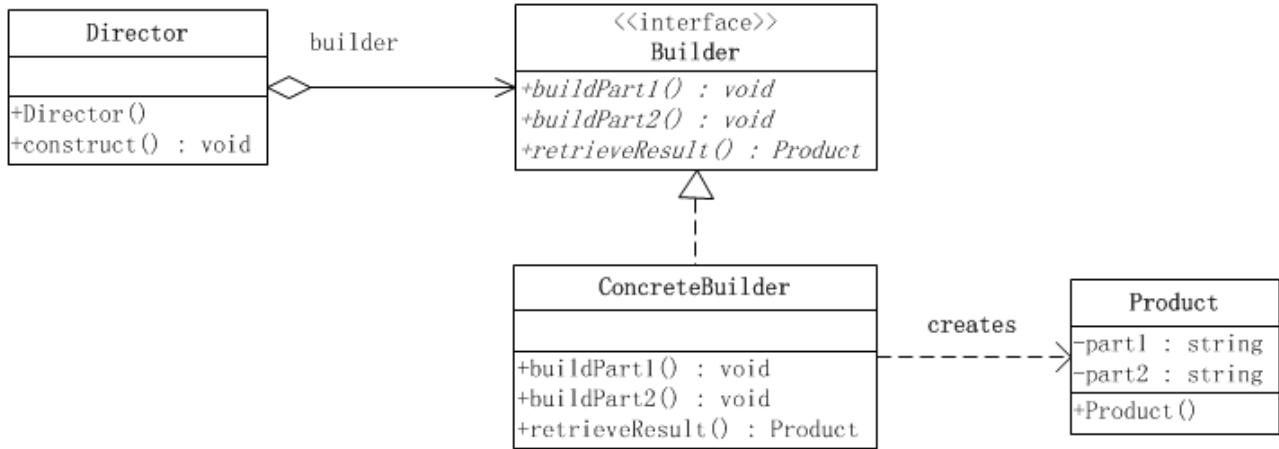
这个模式的优势在于，getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。但若使用反射强制实例化私有方法，那就没办法，所以建议不要再使用反射。

2. 单例和枚举

```
1. public enum Singleton {
2.     /**
3.      * 定义一个枚举的元素，它就代表了Singleton的一个实例。
4.      */
5.
6.     uniqueInstance;
7.
8.     /**
9.      * 单例可以有自己的操作
10.     */
11.    public void singletonOperation(){
12.        //功能处理
13.    }
14. }
```

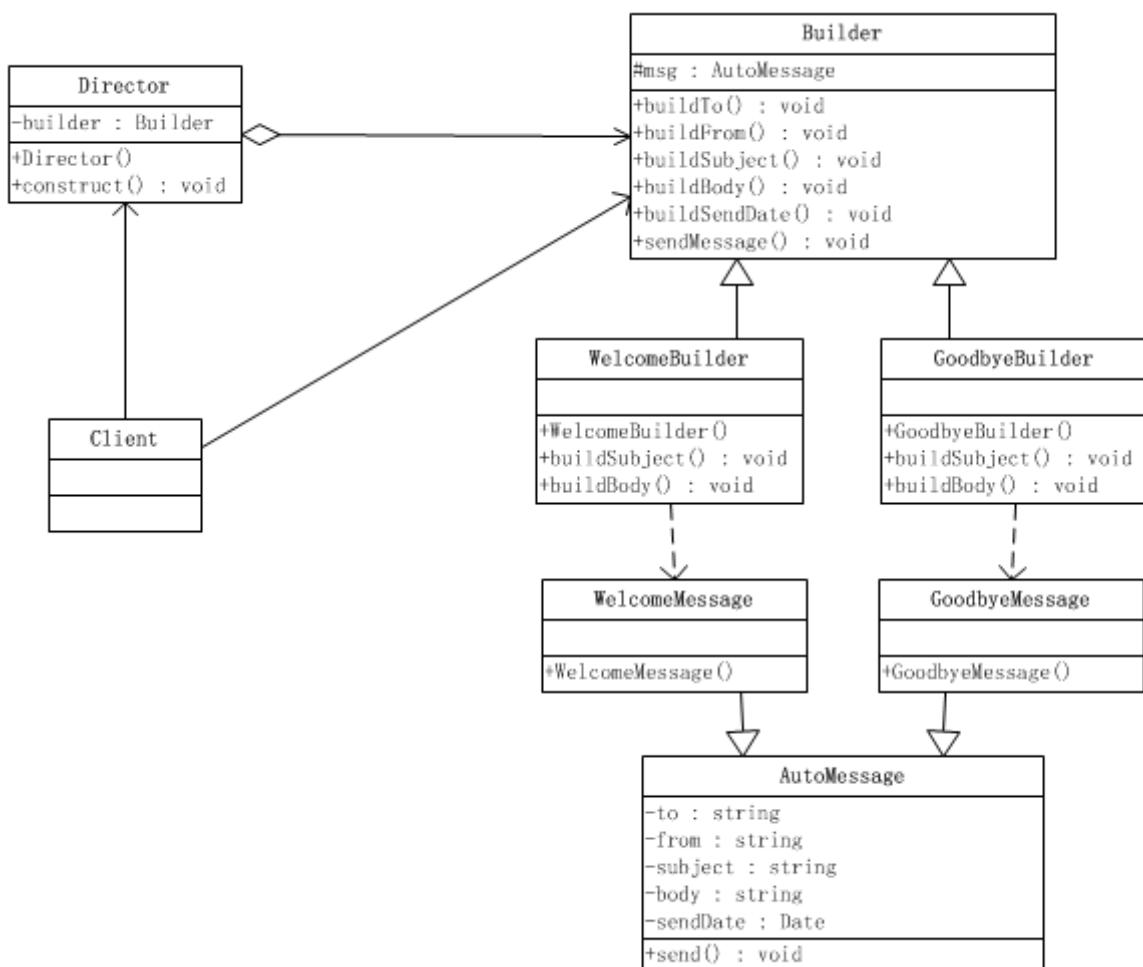
5. 建造模式

建造模式可以将一个产品的内部表象 (internal representation) 与产品的生产过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。



```

1. public class Client {
2.     @Test
3.     public void test() {
4.         Builder builder = new ConcreteBuilder();
5.         Director director = new Director(builder);
6.         director.construct();
7.         Product product = builder.retrieveResult();
8.         System.out.println(product.getPart1());
9.         System.out.println(product.getPart2());
10.    }
11. }
  
```



```

1. public class Director {
2.     Builder builder;
3.     /**
  
```

```
4.     * 构造子
5.     */
6.     public Director(Builder builder) {
7.         this.builder = builder;
8.     }
9.     /**
10.      * 产品构造方法，负责调用各零件的建造方法
11.      */
12.     public void construct(String toAddress, String fromAddress) {
13.         this.builder.buildTo(toAddress);
14.         this.builder.buildFrom(fromAddress);
15.         this.builder.buildSubject();
16.         this.builder.buildBody();
17.         this.builder.buildSendDate();
18.         this.builder.sendMessage();
19.     }
20. }
```

```
1. public class Client {
2.     @Test
3.     public void test() {
4.         Builder builder = new WelcomeBuilder();
5.         Director director = new Director(builder);
6.         director.construct("toAddress@126.com", "fromAddress@126.com");
7.     }
8. }
```

另一种使用场景：

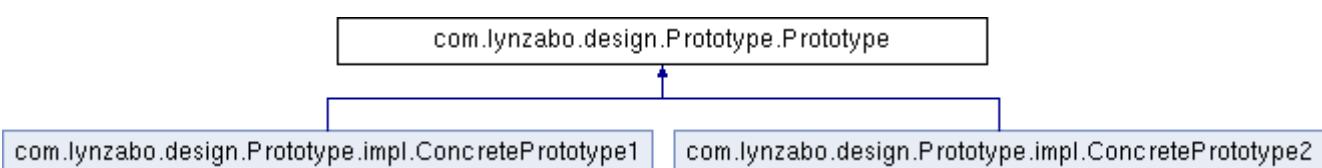
```
1. public class Client {
2.     @Test
3.     public void test() {
4.         //创建构建器对象
5.         InsuranceContract.ConcreteBuilder builder =
6.             new InsuranceContract.ConcreteBuilder("9527", 123L, 456L);
7.         //设置需要的数据，然后构建保险合同对象
8.         InsuranceContract contract =
9.             builder.setPersonName("小明").setOtherData("test").build();
10.        //操作保险合同对象的方法
11.        contract.someOperation();
12.    }
13. }
```

6. 原型模式

通过给出一个原型对象来指明所有创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是选型模式的用意。

原型模式要求对象实现一个可以“克隆”自身的接口，这样就可以通过复制一个实例对象本身来创建一个新的实例。

```
1. public interface Prototype {
2.     public Prototype clone();
3.     public String getName();
4.     public void setName(String name);
5. }
```



```

1. public class PrototypeManager {
2.     /**
3.      * 用来记录原型的编号和原型实例的对应关系
4.      */
5.     private static Map<String, Prototype> map = new HashMap<String, Prototype>();
6.     /**
7.      * 私有化构造方法，避免外部创建实例
8.      */
9.     private PrototypeManager() {
10. }
11. /**
12.      * 向原型管理器里面添加或是修改某个原型注册
13.      *
14.      * @param prototypeId
15.      *          原型编号
16.      * @param prototype
17.      *          原型实例
18.      */
19. public synchronized static void setPrototype(String prototypeId,
20.                                              Prototype prototype) {
21.     map.put(prototypeId, prototype);
22. }
23. /**
24.      * 从原型管理器里面删除某个原型注册
25.      *
26.      * @param prototypeId
27.      *          原型编号
28.      */
29. public synchronized static void removePrototype(String prototypeId) {
30.     map.remove(prototypeId);
31. }
32. /**
33.      * 获取某个原型编号对应的原型实例
34.      *
35.      * @param prototypeId
36.      *          原型编号
37.      * @return 原型编号对应的原型实例
38.      * @throws Exception
39.      *          如果原型编号对应的实例不存在，则抛出异常
40.      */
41. public synchronized static Prototype getPrototype(String prototypeId)
42.         throws Exception {
43.     Prototype prototype = map.get(prototypeId);
44.     if (prototype == null) {
45.         throw new Exception("您希望获取的原型还没有注册或已被销毁");
46.     }
47.     return prototype;
48. }
49. }

```

```

1. public class Client {
2.     @Test
3.     public void test() {
4.         try {

```

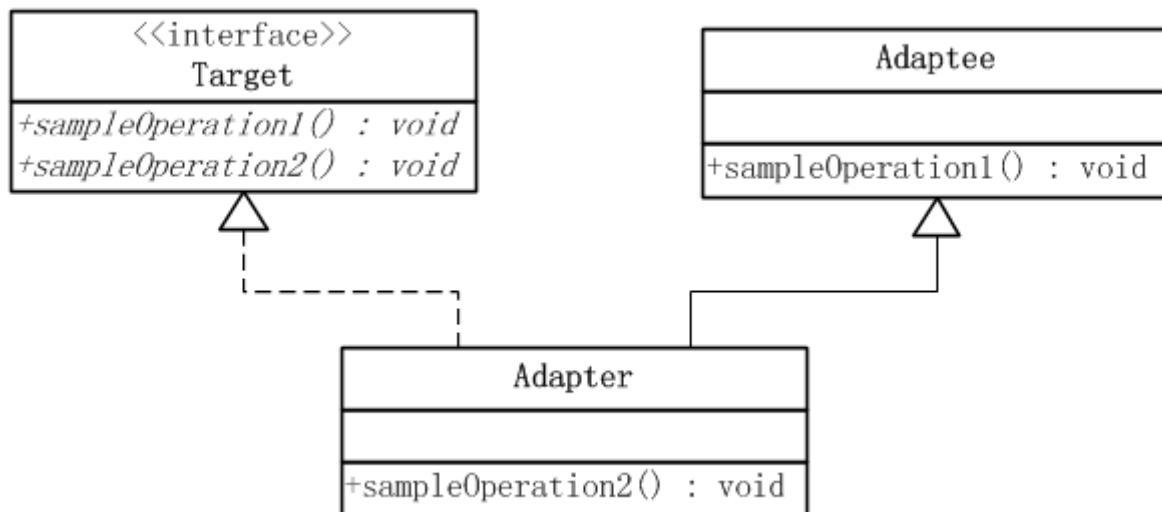
```

5.     Prototype p1 = new ConcretePrototype1();
6.     PrototypeManager.setPrototype("p1", p1);
7.     // 获取原型来创建对象
8.     Prototype p3 = PrototypeManager.getPrototype("p1").clone();
9.     p3.setName("张三");
10.    System.out.println("第一个实例: " + p3);
11.    // 有人动态的切换了实现
12.    Prototype p2 = new ConcretePrototype2();
13.    PrototypeManager.setPrototype("p1", p2);
14.    // 重新获取原型来创建对象
15.    Prototype p4 = PrototypeManager.getPrototype("p1").clone();
16.    p4.setName("李四");
17.    System.out.println("第二个实例: " + p4);
18.    // 有人注销了这个原型
19.    PrototypeManager.removePrototype("p1");
20.    // 再次获取原型来创建对象
21.    Prototype p5 = PrototypeManager.getPrototype("p1").clone();
22.    p5.setName("王五");
23.    System.out.println("第三个实例: " + p5);
24. } catch (Exception e) {
25.     e.printStackTrace();
26. }
27. }
28. }

```

7. 适配器模式

适配器模式把一个类的接口转换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。



```

1. public class Client {
2.     @Test
3.     public void test() {
4.         Target target = new Adapter();
5.         target.sampleOperation1();
6.     }
7. }

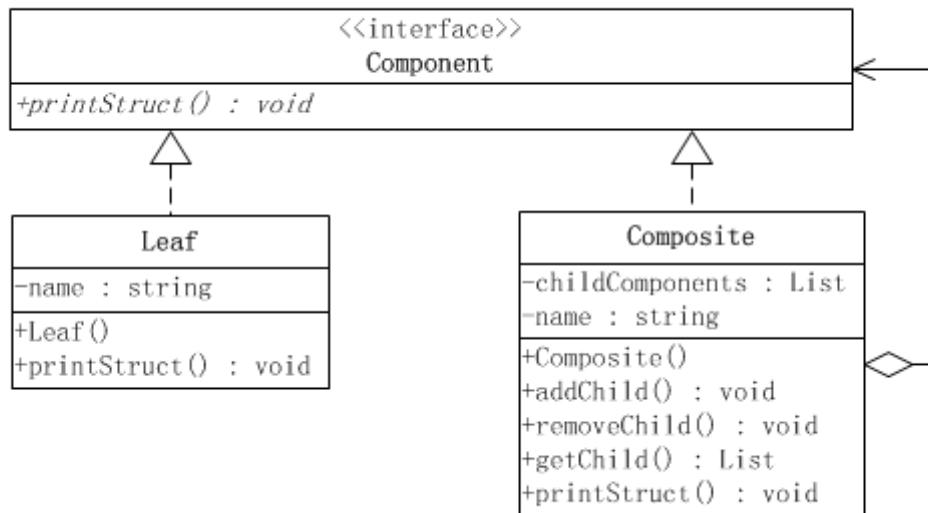
```

在任何时候，如果不准备实现一个接口的所有方法时，就可以使用“缺省适配模式”制造一个抽象类，给出所有方法的平庸的具体实现。这样，从这个抽象类再继承下去的子类就不必实现所有的方法了。

8. 合成模式

合成模式把部分和整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由它们复合而成的合成对象同等看待。

一个文件系统是一个树结构，树上长有节点。树的节点有两种，一种是树枝节点，即目录，有内部树结构，在图中涂有颜色；另一种是文件，即树叶节点，没有内部树结构。

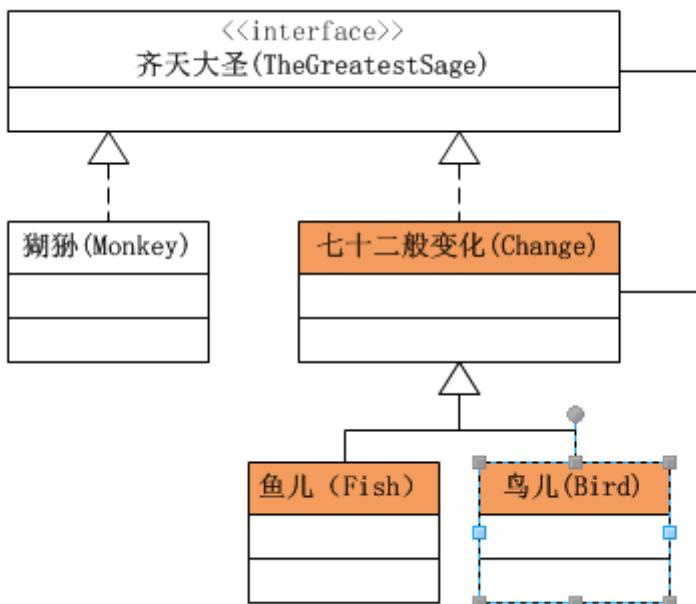
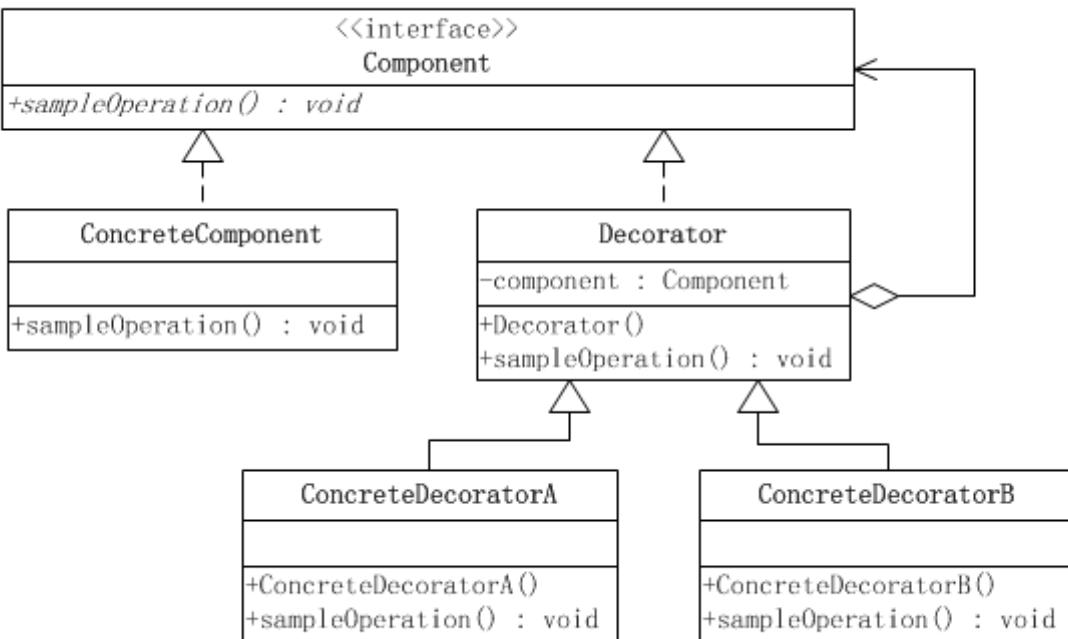


上面树枝和枝干节点都有自己的信息printStruct(), 对于树干，可以添加树枝或树干addChild。

```
1. public class Client {
2.     @Test
3.     public void test() {
4.         Composite root = new Composite("服装");
5.         Composite c1 = new Composite("男装");
6.         Composite c2 = new Composite("女装");
7.         Leaf leaf1 = new Leaf("衬衫");
8.         Leaf leaf2 = new Leaf("夹克");
9.         Leaf leaf3 = new Leaf("裙子");
10.        Leaf leaf4 = new Leaf("套装");
11.        root.addChild(c1);
12.        root.addChild(c2);
13.        c1.addChild(leaf1);
14.        c1.addChild(leaf2);
15.        c2.addChild(leaf3);
16.        c2.addChild(leaf4);
17.        root.printStruct("");
18.    }
19.}
```

9. 装饰模式

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。装饰者模式的一个典型案例就是对输出结果进行增强。比如，现在需要将某一结果通过HTML进行发布，那么首先就需要将内容转化为一个HTML文本。同时，由于内容需要在网络上通过HTTP流传，故，还需要为其增加HTTP头。



```

1. public class Monkey implements TheGreatestSage {
2.     public void move() {
3.         // 代码
4.         System.out.println("Monkey Move");
5.     }
6. }

```

```

1. public class Change implements TheGreatestSage {
2.     private TheGreatestSage sage;
3.     public Change(TheGreatestSage sage) {
4.         this.sage = sage;
5.     }
6.     public void move() {
7.         // 代码
8.         sage.move();
9.     }
10. }

```

```

1. public class Bird extends Change {
2.     public Bird(TheGreatestSage sage) {
3.         super(sage);
4.     }
5.     public void move() {
6.         // 代码
7.         System.out.println("Bird Move");
8.     }
9. }

```

```

1. public class Fish extends Change {
2.     public Fish(TheGreatestSage sage) {
3.         super(sage);
4.     }
5.     public void move() {
6.         // 代码
7.         System.out.println("Fish Move");
8.     }
9. }

```

```

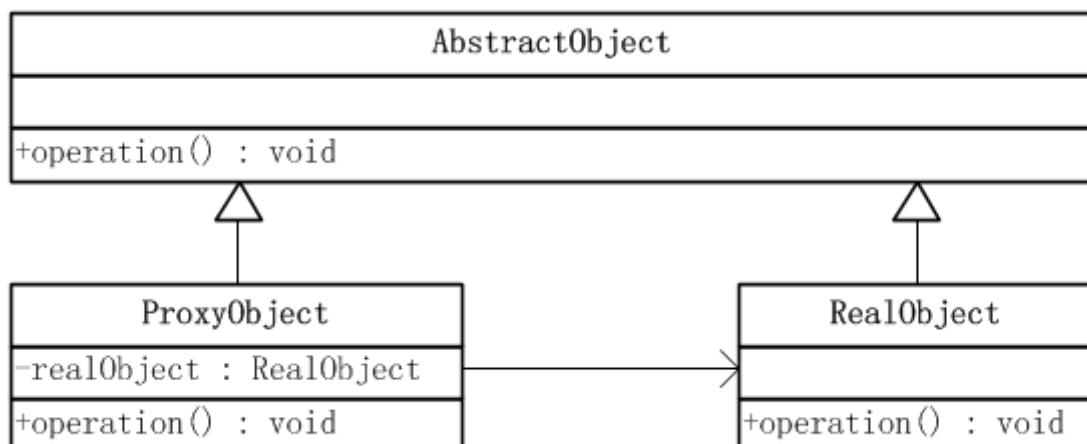
1. public class Client {
2.     @Test
3.     public void test(){
4.         TheGreatestSage sage = new Monkey();
5.         // 第一种写法
6.         TheGreatestSage bird = new Bird(sage);
7.         TheGreatestSage fish = new Fish(bird);
8.         // 第二种写法
9.         //TheGreatestSage fish = new Fish(new Bird(sage));
10.        fish.move();
11.    }
12. }

```

10. 代理模式

代理模式是使用代理对象完成用户请求，屏蔽用户对真实对象的访问。

使用代理模式的意图很多，比如因为安全原因，需要屏蔽客户端直接访问真实对象；或者在远程调用中，需要使用代理类处理远程方法调用的技术细节（如RMI、Dubbo）。Spring AOP用的很多。



```

1. /**
2.  * 目标对象角色

```

```
3. * @author Lynzabo
4.
5. */
6. public class DBQuery extends IDBQuery {
7.     public void operation() {
8.         //一些操作
9.         System.out.println("一些操作");
10.    }
11. }
```

```
1. /**
2.  * 代理对象角色
3. *
4. * @author Lynzabo
5. *
6. */
7. public class DBQueryProxy extends IDBQuery {
8.     DBQuery realObject = new DBQuery();
9.     public void operation() {
10.         // 调用目标对象之前可以做相关操作
11.         System.out.println("before");
12.         realObject.operation();
13.         // 调用目标对象之后可以做相关操作
14.         System.out.println("after");
15.     }
16. }
```

```
1. public class Client {
2.     @Test
3.     public void test() {
4.         IDBQuery obj = new DBQueryProxy();
5.         obj.operation();
6.     }
7. }
```

Java动态代理

动态代理是指在运行时，动态生成代理类。即，**代理类的字节码将在运行时生成并载入当前的ClassLoader**。

生成动态代理类的方法很多，如，JDK自带的动态代理、CGLIB、Javassist或者ASM库。

- JDK的动态代理使用简单，它内置在JDK中，因此不需要引入第三方Jar包，但相对功能比较弱，**只能对实现了接口的类生成代理，而不能针对类**。
- CGLIB和Javassist（提供代理工厂和使用动态代码创建，代理工厂和CGLIB差不多，使用动态代码是可以在代理类代码中写类的java代码，比如指定给类写一个方法然后让生效）都是高级的字节码生成库，总体性能比JDK自带的动态代理好，而且功能十分强大，**是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法**。**因为是继承，所以该类或方法最好不要声明成final，final可以阻止继承和多态。**
- ASM是低级的字节码生成工具，使用ASM已经近乎于在使用Java bytecode编程，对开发人员要求最高。当前，也是性能最好的一种动态代理生成工具。CGLIB以ASM为基础，对ASM的功能进行了扩展和封装，提供了更友好的API。

但ASM的使用实在过于繁琐，而且性能也没有数量级的提升，与CGLIB等高级字节码生成工具相比，ASM程序的可维护性也较差，如果不是在对性能有苛刻要求的场合，笔者还是推荐CGLIB或者Javassist。就不要看Javassist了。

JDK：

```
1. public Object newProxy(Object targetObject) { // 将目标对象传入进行代理
```

```
2.     this.targetObject = targetObject; <br>      //注意这个方法的参数，后面是类实现的接口
3.     return Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),
4.                                    targetObject.getClass().getInterfaces(), this); // 返回代理对象
5. }
```

在生成代理类时，传递的是实现类所实现的接口 **targetObject.getClass().getInterfaces()**，所以 **JDK** 只能对于接口进行做代理。

CGLIB：

```
1. public Object createProxyObject(Object obj) {
2.     this.targetObject = obj;
3.     Enhancer enhancer = new Enhancer();
4.     enhancer.setSuperclass(obj.getClass());
5.     enhancer.setCallback(this);
6.     Object proxyObj = enhancer.create();
7.     return proxyObj; // 返回代理对象，返回的对象其实就是一个封装了“实现类”的代理类，是实现类的实例。
8. }
```

AOP (Aspect-Oriented Programming, 面向切面编程)，AOP包括切面 (aspect)、通知 (advice)、连接点 (joinpoint)，实现方式就是通过对目标对象的代理在连接点前后加入通知，完成统一的切面操作。

实现AOP的技术，主要分为两大类：

一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring提供了两种方式来生成代理对象: **JDKProxy**和**Cglib**，具体使用哪种方式生成由**AopProxyFactory**根据**AdvisedSupport**对象的配置来决定。

默认的策略是如果目标类是接口，则使用JDK动态代理技术，如果目标对象没有实现接口，则默认会采用CGLIB代理。

如果目标对象实现了接口，可以强制使用CGLIB实现代理（添加CGLIB库，并在spring配置中加入[`<aop:aspectj-autoproxy proxy-target-class="true"/>`](#)）。

在mybatis中，使用了很多动态代理，非常的普遍，使用最多的是jdk proxy，然后就是懒加载用到的cglib（默认）和javassist。Hibernate也类似。

启动懒加载，mybatis初始化返回类型的时候，会返回一个cglib代理对象，cglib对象会过滤get, set, is, "equals", "clone", "hashCode", "toString"触发方法，当调用这些方法时候，才去查询数据库。

不启动懒加载，不会返回代理对象，返回原生对象，然后会在一开始的时候就加载关联对象和sql中指定的所有属性。

首先，懒加载会用到这两个配置

```
1. <!-- 查询时，关闭关联对象即时加载以提高性能 -->
2. <setting name="lazyLoadingEnabled" value="true" />
3. <!-- 设置关联对象加载的形态，此处为按需加载字段(加载字段由SQL指定)，不会加载关联表的所有字段，以提高性能
4. <setting name="aggressiveLazyLoading" value="false" />
```

这两个配置，第一个毫无疑问，就是启动懒加载的；第二个呢，是mybatis执行完sql语句，组装返回的对象的时候，按sql指定的字段来加载，详细点说，就是如果没有调用指定的字段或者对象的get, set, is, "equals", "clone", "hashCode", "toString"方法，该属性或者关联对象就不会在返回的对象结果中。下面是Hibernate的，mybatis类似。

```
1. //从数据库载入ID为1的用户
2. User u = (User) HibernateSessionFactory.getSession().load(User.class, 1);
3. //打印类名称
4. System.out.println("Class Name:" + u.getClass().getName()); // Class Name:$javatuning.ch2
```

```

5. //打印父类名称
6. System.out.println("Super Class Name:"+u.getClass().getSuperclass().getName());//Super Cl
7. //实现的所有接口
8. Class[] ins = u.getClass().getInterfaces();
9. for(Class cls:ins) {
10.     System.out.println("interface:"+cls.getName());//org.hibernate.proxy.HibernateProxy
11. }
12. System.out.println(u.getName());//Hibernate:select user0_.id as id0_0_,user0_.name as nam

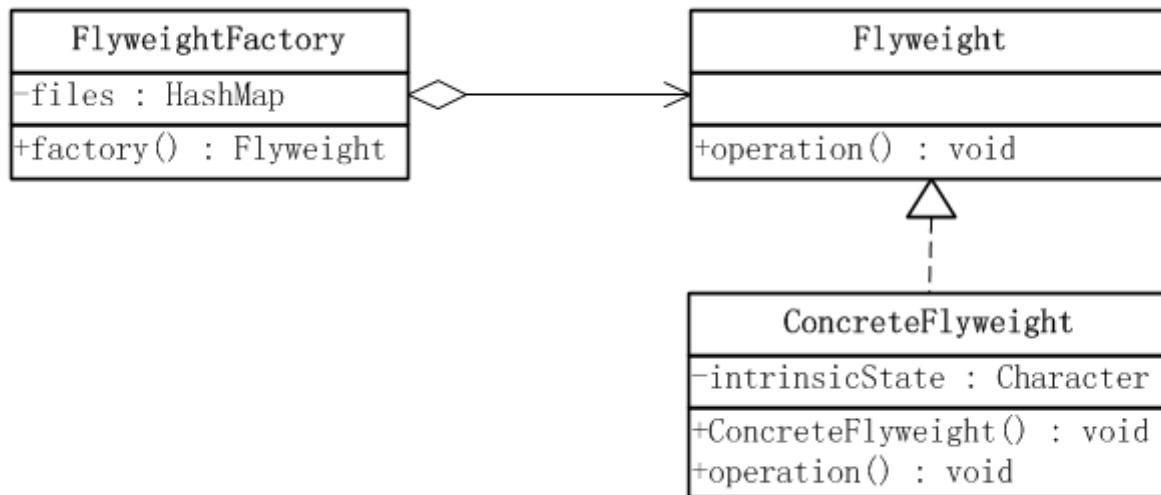
```

由这段输出来看，在getName()被调用之前，Hibernate从未输出过一条SQL语句。这表示，User对象呗加载时，根本没有访问数据库，而在getName()方法被调用时，才真正完成了数据库操作。

TODO 看JVM时候，详细看.java文件到jvm的过程，在运行期间产生字节码。利用字节码增强，动态生成.class文件。

11. 享元模式

享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。



```

1. public class FlyweightFactory {
2.     private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();
3.     public Flyweight factory(Character state) {
4.         // 先从缓存中查找对象
5.         Flyweight fly = files.get(state);
6.         if (fly == null) {
7.             // 如果对象不存在则创建一个新的Flyweight对象
8.             fly = new ConcreteFlyweight(state);
9.             // 把这个新的Flyweight对象添加到缓存中
10.            files.put(state, fly);
11.        }
12.        return fly;
13.    }
14. }

```

```

1. public class Client {
2.     @Test
3.     public void test() {
4.         FlyweightFactory factory = new FlyweightFactory();
5.         Flyweight fly = factory.factory(new Character('a'));

```

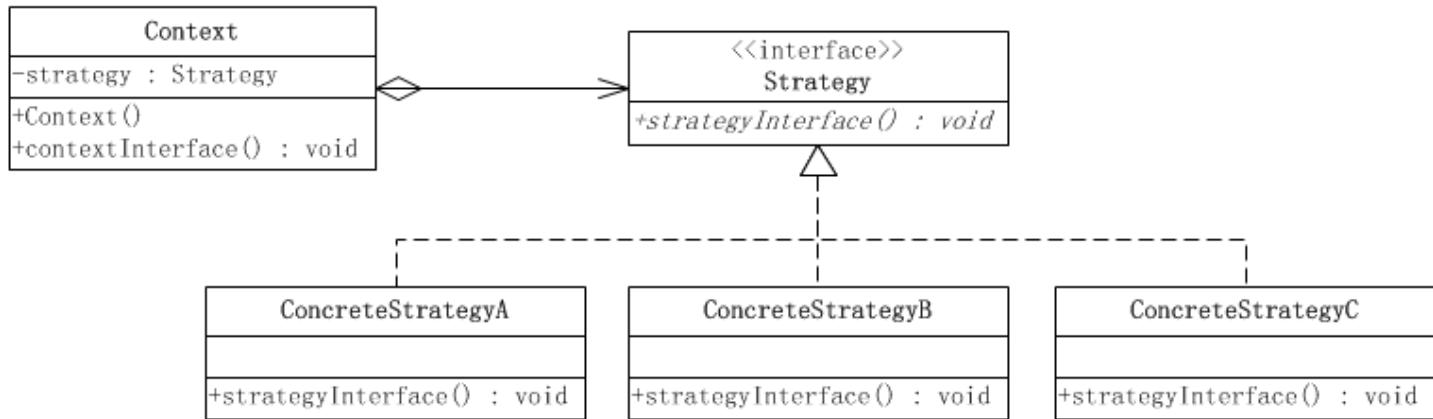
```

6.         fly.operation("First Call");
7.         fly = factory.factory(new Character('b'));
8.         fly.operation("Second Call");
9.         fly = factory.factory(new Character('a'));
10.        fly.operation("Third Call");
11.    }
12.}

```

12. 策略模式

策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。



```

1. public class PrimaryMemberStrategy implements MemberStrategy {
2.     public double calcPrice(double booksPrice) {
3.         System.out.println("对于初级会员的没有折扣");
4.         return booksPrice;
5.     }
6. }
7. public class IntermediateMemberStrategy implements MemberStrategy {
8.     public double calcPrice(double booksPrice) {
9.         System.out.println("对于中级会员的折扣为10%");
10.        return booksPrice * 0.9;
11.    }
12. }
13. public class AdvancedMemberStrategy implements MemberStrategy {
14.     public double calcPrice(double booksPrice) {
15.         System.out.println("对于高级会员的折扣为20%");
16.         return booksPrice * 0.8;
17.     }
18. }
19. public class Price {
20.     // 持有一个具体的策略对象
21.     private MemberStrategy strategy;
22.
23.     /**
24.      * 构造函数，传入一个具体的策略对象
25.      *
26.      * @param strategy
27.      *          具体的策略对象
28.      */
29.     public Price(MemberStrategy strategy) {
30.         this.strategy = strategy;

```

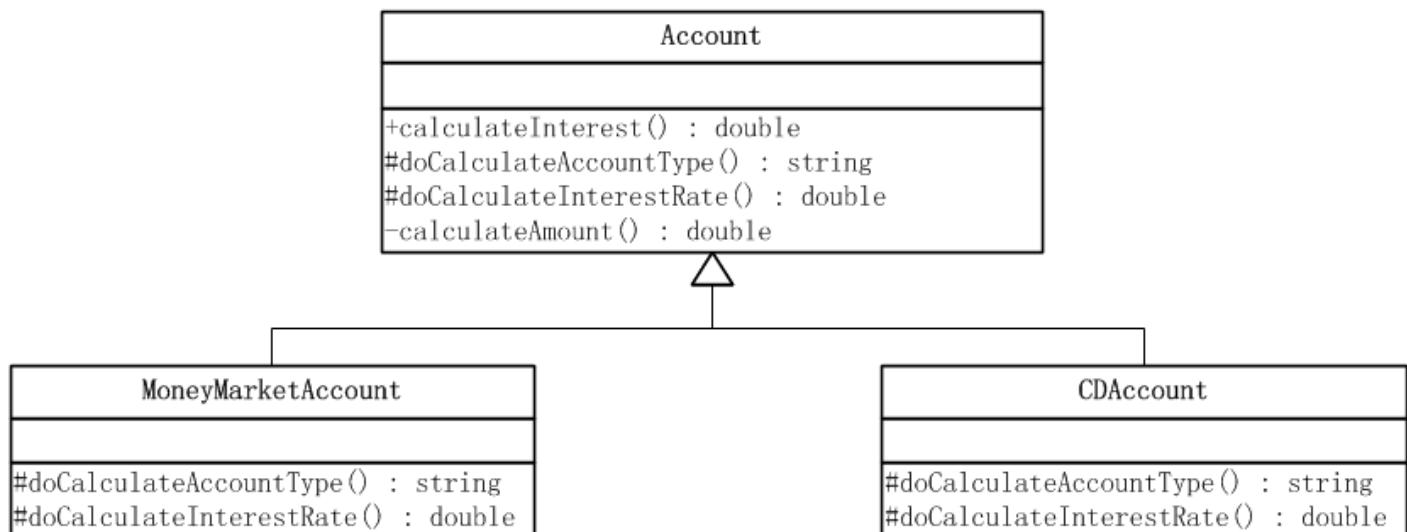
```

31. }
32. /**
33. * 计算图书的价格
34. *
35. * @param booksPrice
36. *          图书的原价
37. * @return 计算出打折后的价格
38. */
39. public double quote(double booksPrice) {
40.     return this.strategy.calcPrice(booksPrice);
41. }
42. }
43. public class Client {
44.     @Test
45.     public void test() {
46.         // 选择并创建需要使用的策略对象
47.         MemberStrategy strategy = new AdvancedMemberStrategy();
48.         // 创建环境
49.         Price price = new Price(strategy);
50.         // 计算价格
51.         double quote = price.quote(300);
52.         System.out.println("图书的最终价格为: " + quote);
53.     }
54. }

```

13. 模板方法模式

模板方法模式是类的行为模式。可以声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。



```

1. public abstract class Account {
2.     /**
3.      * 模板方法，计算利息数额
4.      *
5.      * @return 返回利息数额
6.      */
7.     public final double calculateInterest() {
8.         double interestRate = doCalculateInterestRate();
9.         String accountType = doCalculateAccountType();
10.        double amount = calculateAmount(accountType);
11.        return amount * interestRate;
12.    }

```

```

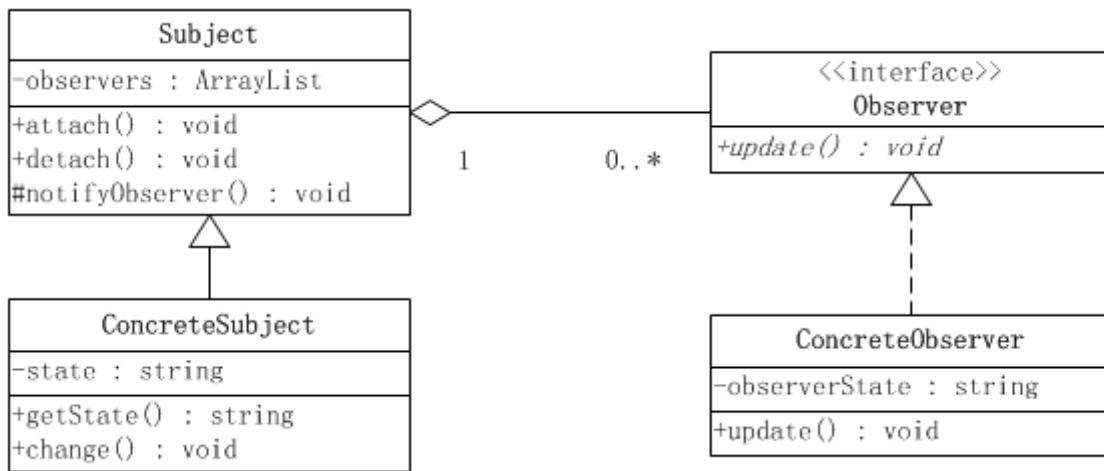
13. /**
14. * 基本方法留给子类实现
15. */
16. protected abstract String doCalculateAccountType();
17. /**
18. * 基本方法留给子类实现
19. */
20. protected abstract double doCalculateInterestRate();
21. /**
22. * 基本方法, 已经实现
23. */
24. private double calculateAmount(String accountType) {
25. /**
26. * 省略相关的业务逻辑
27. */
28. return 7243.00;
29. }
30. }
31. public class MoneyMarketAccount extends Account {
32. @Override
33. protected String doCalculateAccountType() {
34.     return "Money Market";
35. }
36. @Override
37. protected double doCalculateInterestRate() {
38.     return 0.045;
39. }
40. }
41. public class CDAccount extends Account {
42. @Override
43. protected String doCalculateAccountType() {
44.     return "Certificate of Deposite";
45. }
46. @Override
47. protected double doCalculateInterestRate() {
48.     return 0.06;
49. }
50. }
51. public class Client {
52. @Test
53. public void test() {
54.     Account account = new MoneyMarketAccount();
55.     System.out.println("货币市场账号的利息数额为: " + account.calculateInterest());
56.     account = new CDAccount();
57.     System.out.println("定期账号的利息数额为: " + account.calculateInterest());
58. }
59. }

```

HttpServlet中的doGet/doPost就用了模板方法模式

14. 观察者模式

观察者模式是对象的行为模式，又叫发布-订阅(Publish/Subscribe)模式。



推模型和拉模型

在观察者模式中，又分为推模型和拉模型两种方式。

- **推模型**

主题对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。

- **拉模型**

主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中获取。

一般这种模型的实现中，会把主题对象自身通过**update()**方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

推模型

```

1. public abstract class Subject {
2.     /**
3.      * 用来保存注册的观察者对象
4.      */
5.     private List<Observer> list = new ArrayList<Observer>();
6.     /**
7.      * 注册观察者对象
8.      *
9.      * @param observer
10.     *          观察者对象
11.     */
12.    public void attach(Observer observer) {
13.        list.add(observer);
14.        System.out.println("Attached an observer");
15.    }
16.    /**
17.     * 删除观察者对象
18.     *
19.     * @param observer
20.     *          观察者对象
21.     */
22.    public void detach(Observer observer) {
23.        list.remove(observer);
24.    }
25.    /**
26.     * 通知所有注册的观察者对象
27.     */
28.    public void notifyObservers(String newState) {
29.        for (Observer observer : list) {
30.            observer.update(newState);
31.        }
32.    }
33. }

```

```

34. public class ConcreteSubject extends Subject {
35.     private String state;
36.     public String getState() {
37.         return state;
38.     }
39.     public void change(String newState) {
40.         state = newState;
41.         System.out.println("主题状态为: " + state);
42.         // 状态发生改变, 通知各个观察者
43.         this.notifyObservers(state);
44.     }
45. }
46. public interface Observer {
47.     /**
48.      * 更新接口
49.      *
50.      * @param state
51.      *          更新的状态
52.      */
53.     public void update(String state);
54. }
55. public class ConcreteObserver implements Observer {
56.     // 观察者的状态
57.     private String observerState;
58.     public void update(String state) {
59.         /**
60.          * 更新观察者的状态, 使其与目标的状态保持一致
61.          */
62.         observerState = state;
63.         System.out.println("状态为: " + observerState);
64.     }
65. }
66. public class Client {
67.     @Test
68.     public void test() {
69.         // 创建主题对象
70.         ConcreteSubject subject = new ConcreteSubject();
71.         // 创建观察者对象
72.         Observer observer = new ConcreteObserver();
73.         // 将观察者对象登记到主题对象上
74.         subject.attach(observer);
75.         // 改变主题对象的状态
76.         subject.change("new state");
77.     }
78. }

```

拉模型

```

1. public interface Observer {
2.     /**
3.      * 更新接口
4.      *
5.      * @param subject
6.      *          传入主题对象, 方便获取相应的主题对象的状态
7.      */
8.     public void update(Subject subject);
9. }
10. public class ConcreteObserver implements Observer {
11.     // 观察者的状态
12.     private String observerState;
13.     public void update(Subject subject) {

```

```

14.         /**
15.          * 更新观察者的状态，使其与目标的状态保持一致
16.          */
17.         observerState = ((ConcreteSubject) subject).getState();
18.         System.out.println("观察者状态为: " + observerState);
19.     }
20. }
21. public abstract class Subject {
22.     /**
23.      * 用来保存注册的观察者对象
24.      */
25.     private List<Observer> list = new ArrayList<Observer>();
26.     /**
27.      * 注册观察者对象
28.      *
29.      * @param observer
30.      *          观察者对象
31.      */
32.     public void attach(Observer observer) {
33.         list.add(observer);
34.         System.out.println("Attached an observer");
35.     }
36.     /**
37.      * 删除观察者对象
38.      *
39.      * @param observer
40.      *          观察者对象
41.      */
42.     public void detach(Observer observer) {
43.         list.remove(observer);
44.     }
45.     /**
46.      * 通知所有注册的观察者对象
47.      */
48.     public void notifyObservers() {
49.         for (Observer observer : list) {
50.             observer.update(this);
51.         }
52.     }
53. }
54. public class ConcreteSubject extends Subject {
55.     private String state;
56.     public String getState() {
57.         return state;
58.     }
59.     public void change(String newState) {
60.         state = newState;
61.         System.out.println("主题状态为: " + state);
62.         // 状态发生改变，通知各个观察者
63.         this.notifyObservers();
64.     }
65. }

```

java本身对observe提供了支持，提供java.util.Observable类。

15. 门面模式

对下面显示都隐藏，只对外暴露几个核心接口。如我做过的工作流引擎，只对外暴露start()、pause()等方法，而将底下的实现都屏蔽了。Facade。

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

Java BIO与NIO、AIO

BIO是面向流传输，同步阻塞的IO；NIO和AIO是面向缓冲的，NIO是同步非阻塞的IO，AIO是异步非阻塞的IO。

名词解释：

- 同步 指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪
- 异步 异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知（异步的特点就是通知）
- 阻塞 所谓阻塞方式的意思是指，当试图对该文件描述符进行读写时，如果当时没有东西可读，或者暂时不可写，程序就进入等待状态，直到有东西可读或者可写为止。
- 非阻塞 非阻塞状态下，如果没有东西可读，或者不可写，读写函数马上返回，而不会等待，

下面我们再来理解组合方式的IO类型：

- 同步阻塞IO (JAVA BIO)：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
- 同步非阻塞IO(Java NIO)：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问。
- 异步阻塞IO (Java NIO)：此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄（如果从UNP的角度看，select属于同步操作。因为select之后，进程还需要读写数据），从而提高系统的并发性！
- 异步非阻塞IO (Java AIO(NIO.2))：在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

BIO、NIO、AIO适用场景分析：

- BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
- AIO方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

在IO设计中，我们从InputStream或Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

1. Name: Anna
2. Age: 25
3. Email: anna@mailserver.com
4. Phone: 1234567890

该文本行的流可以这样处理：

```
1. InputStream input = ... ; // get the InputStream from the client socket
2. BufferedReader reader = new BufferedReader(new InputStreamReader(input));
3. String nameLine = reader.readLine();
4. String ageLine = reader.readLine();
5. String emailLine = reader.readLine();
6. String phoneLine = reader.readLine();
```

处理状态由程序执行多久决定。换句话说，一旦`reader.readLine()`方法返回，你就知道肯定文本行就已读完，`readline()`阻塞直到整行读完。正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。

而一个NIO的实现会有所不同，下面是一个简单的例子：

```
1. ByteBuffer buffer = ByteBuffer.allocate(48);
2. int bytesRead = inChannel.read(buffer);
```

注意第二行，从通道读取字节到`ByteBuffer`。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。

假设第一次`read(buffer)`调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

```
1. ByteBuffer buffer = ByteBuffer.allocate(48);
2. int bytesRead = inChannel.read(buffer);
3. while(!bufferFull(bytesRead) ) {
4.     bytesRead = inChannel.read(buffer);
5. }
```

`bufferFull()`方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。我们对NIO处理做进一步的优化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

AIO与NIO不同，当进行读写操作时，只须直接调用API的`read`或`write`方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入`read`方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将`write`方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，`read/write`方法都是异步的，完成后会主动调用回调函数。在JDK1.7中，这部分内容被称作NIO.2，主要在`java.nio.channels`包下提供几种异步通道。

select、epoll、kqueue、ioCP I/O复用机制

介绍几种常见的I/O模型及其区别:

1. blocking I/O(阻塞套接字)

应用程序请求内核获取数据，一直阻塞（阻塞过程中包含等待数据，将数据从内核空间拷贝到用户空间），直到内核返回数据。

2.nonblocking I/O(非阻塞套接字)

内核提供返回数据接口，若无数据，返回空，否则返回数据。

应用程序轮询请求内核获取数据接口，直到内核有数据，然后阻塞将数据从内核中拷贝出来。

3.I/O multiplexing(I/O复用模型select and poll)

可以同时处理多个套接字。

select先阻塞，有活动套接字才返回。与blocking I/O相比，select会有两次系统调用（有活动套接字和将数据从内核空间拷贝到用户空间），但是select能处理多个套接字。

4.signal driven I/O (SIGIO)

免去了select的阻塞与轮询，当有活跃套接字时，由注册的handler(回调函数)处理，也需要阻塞从内核空间拷贝数据到用户控件。

5.asynchronous I/O (完全异步的I/O复用机制the POSIX aio_functions)

纵观上面其它四种模型，至少都会在由kernel copy data to application时阻塞。而该模型是当copy完成后才通知application，可见是纯异步的。好像只有windows的完成端口是这个模型，效率也很出色。

很少有*nix系统支持，windows的IOCP则是此模型

可以看出，越往后，阻塞越少，理论上效率也是最优。

select和ioCP分别对应第3种与第5种模型，epoll与kqueue其实也于select属于同一种模型，只是可以看作有了第4种模型的某些特性，如callback机制。

阻塞I/O模式下，一个线程只能处理一个流的I/O事件，内核对于I/O事件的处理是阻塞或者唤醒。如果想要同时处理多个流，要么多进程(fork)，要么多线程(pthread_create)，很不幸这两种方法效率都不高。传统做法是，如果我们想要同时处理多个流的I/O事件，可以不停的把所有流从头到尾问一遍，一直循环问，但这样的做法显然不好，因为如果所有的流都没有数据，那么只会白白浪费CPU。这就是非阻塞忙轮询的I/O方式，在非阻塞模式下，想同时处理多个流，则可以把I/O事件交给select、epoll等来处理，使用它们可以同时观察多个流的I/O事件，，例如使用select，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有I/O事件时，就从阻塞态中醒来，这时我们的程序再轮询一遍所有的流，读出数据，这时就不是忙轮询所有流。如果没有I/O事件产生，我们的程序就会阻塞在select处。但是依然有个问题，我们从select那里仅仅知道了，有I/O事件发生了，但却并不知道是那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。这时来介绍epoll，epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们，此时我们对这些流的操作都是有意义的，epoll_wait函数等待直到注册的事件发生。

epoll的原理就是把要监控读写的文件交给内核(epoll_add)，添加/删除你关心的事件(epoll_ctl)，比如读事件，然后等(epoll_wait)，此时，如果没有哪个文件有你关心的事件，则休眠，直到有事件，被唤醒然后返回那些事件

为什么epoll,kqueue比select高级？

epoll也是轮询，只不过是对产生了io事件的socket进行轮询，然后直接调用callback，而不需要像select或poll那样对所有的socket都进行轮询，不管哪个socket是活跃的。这会浪费很多CPU时间。同时也减少对文件句柄的拷

贝。

我们在调用epoll_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件。当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll_wait非常高效。当我们执行epoll_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。而select将所有文件描述符传递给内核中，内核处理读、写缓冲区，内核在检测到文件句柄可读/可写时就产生中断通知监控者select，select被内核触发之后，就返回可读可写的文件句柄的总数；内核会修改这些文件句柄，所以再次监控时，又需要重新复制一遍原始的文件描述符信息给内核。如此每次都要进行文件句柄拷贝。

2.减少对可读可写文件句柄的遍历；

select/poll/epoll最终总结比较：

select模型

1. 最大并发数限制，因为一个进程所打开的FD（文件描述符）是有限制的，由FD_SETSIZE设置，默认值是1024/2048，因此Select模型的最大并发数就被相应限制了。
2. 效率问题，select每次调用都会线性扫描全部的FD集合，这样效率就会呈现线性下降。
3. 内核/用户空间内存拷贝问题，如何让内核把FD消息通知给用户空间呢？在这个问题上select采取了内存拷贝方法。

poll模型

基本上效率和select是相同的，select缺点的2和3它都没有改掉。

Epoll的模型

select缺点就是epoll有点。

1. Epoll没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于2048，一般来说这个数目和系统内存关系很大。
2. 效率提升，Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。
3. 内存拷贝，Epoll在这点上使用了“共享内存”，这个内存拷贝也省略了。

epoll, kqueue是Reactor模式，IOCP是Proactor模式。

java.nio包是select模型。

Java静态块、静态方法、非静态方法、构造方法执行次序

```
1. public class ExA {  
2.     static {  
3.         System.out.println("父类--静态代码块");  
4.     }  
5.     public ExA() {  
6.         System.out.println("父类--构造函数");  
7.     }  
8.     {  
9.         System.out.println("父类--非静态代码块");  
10.    }  
11.    public void test() {  
12.    }  
13.}
```

```
12.         System.out.println("父类--方法"); //被子类的相同方法覆盖
13.     }
14. }
```

```
1. public class ExB extends ExA {
2.     static {
3.         System.out.println("子类--静态代码块");
4.     }
5.     public ExB() {
6.         System.out.println("子类--构造函数");
7.     }
8.     {
9.         System.out.println("子类--非静态代码块");
10.    }
11.    public void test() {
12.        System.out.println("子类--方法");
13.    }
14.    public static void main(String[] args) {
15.        //new ExB();
16.        //ExB.test();
17.    }
18. }
```

情况一：main方法内都注释掉，执行结果：

- 1. 父类--静态代码块
- 2. 子类--静态代码块

当启动Application，JVM会先加载所有类的静态变量、静态方法、静态代码块，执行静态代码块。

情况二：只打开new ExB()注释，执行结果：

- 1. 父类--静态代码块
- 2. 子类--静态代码块
- 3. 父类--非静态代码块
- 4. 父类--构造函数
- 5. 子类--非静态代码块
- 6. 子类--构造函数
- 7. 子类--方法

当实例化对象时，先执行该类的非静态代码块，然后执行构造方法。

情况三：只打开ExB.test()注释，执行结果：

- 1. 父类--静态代码块
- 2. 子类--静态代码块
- 3. 子类--静态方法

当执行静态方法时，由于不需要实例化对象，所以不会执行非静态代码块、构造方法。

JVM

1. 概述

最有可能成为系统瓶颈的计算资源如下：

- 磁盘I/O：磁盘I/O读写的速度要比内存慢很多，程序在运行过程中，如果需要等待磁盘I/O完成。
- 网络操作：对网络数据进行读写的情况与磁盘I/O类似。
- CPU：对计算资源要求较高的应用，由于其长时间、不间断地大量占用CPU资源，那么对CPU的争夺将导致性能问题。
- 异常：对Java应用来说，异常的捕获和处理是非常消耗资源的。
- 数据库：大部分应用程序都离不开数据库，而应用程序可能需要等待数据库操作完成或者返回请求的结果集。
- 锁竞争：对高并发程序来说，激烈的锁竞争将会明显增加线程上下文切换的开销。
- 内存：一般来说，只要应用程序设计合理，内存读写速度上不太可能成为性能瓶颈。除非内存大小不足。

2. 性能调优的层次

设计调优、代码调优、JVM调优、数据库调优、操作系统调优。

2.1 设计调优

1 StringBuffer与StringBuilder

由于String对象是不可变对象，因此，在需要对字符串进行修改操作时（如字符串连接、替换），String对象总是会生成新的对象，所以，其性能相对较差。为此，JDK专门提供了用于创建和修改字符串的工具，这就是StringBuffer和StringBuilder类。

一旦String对象实例生成，就不可能再被改变。如String result = "String"+ "end" + "String" + "append";首先，由"String"和"end"两个字符串生成"Stringend"对象，然后依次生成"StringendString"和"StringendStringappend"对象。

对于上面的静态字符串的连接操作，Java在编译时会进行彻底的优化，将多个连接操作的字符串在编译时合并成一个单独的长字符串。反而使用StringBuilder性能却没有静态字符串连加性能好。

String加法操作虽然会被优化，但编译器没有那么聪明，所以对于String操作，尽量少用。

StringBuffer和StringBuilder区别：

StringBuffer对几乎所有的方法做了同步，而StringBufler几乎没有，在无需考虑线程安全的情况下可以使用性能相对较好的StringBuilder，但若系统有线程安全要求，只能选择StringBuffer。

StringBuffer和StringBuilder使用二者时注意：

无论是StringBuffer还是StringBuilder，初始化时可以设置一个容量参数，默认是16个字节。数组大小为16。在追加字符串时，如果需要容量超过实际char数组长度，数组会扩容，扩容步骤：

```
1. int newCapacity = (value.length + 1) * 2;           //容量翻倍
2. value = Arrays.copyOf(value, newCapacity);           //将原数组内容重新复制到重新创建的newCapacity大小的
```

所以，如果能够预选评估StringBuilder或StringBuffer的大小，将能够有效地节省这些操作，从而提高系统的性能。

2 Collection

list接口

list接口的实现有3种：ArrayList、Vector和LinkedList。

ArrayList和Vector使用了数组实现，几乎使用了相同的算法，它们的唯一区别是对多线程的支持。ArrayList没有对任何一个方法做线程同步，因此不是线程安全的。Vector中绝大部分方法都做了线程同步，是一种线程安全的实

现。

LinkedList使用了循环双向链表数据结构。LinkedList链表由一系列表项连接而成。一个表项总是包含3个部分：元素内容、前驱表项和后驱表项。

只有当ArrayList的容量大小超过当前数组大小时，才需要扩容。扩容大小=原始容量*1.5倍。新创建一个扩容大小的数组，将旧数组所有元素重新拷贝到新的数组中。因此，合理的数组大小有助于减少数组扩容的次数，从而提高系统性能。

默认情况下，ArrayList数组的初始值大小为10，每次扩容新数组大小设置成原大小的1.5倍。

基于数组的，在任意位置插入或删除元素性能相对低下，在数组的任意位置插入元素，必须导致在该位置后的所有元素需要重新排列，元素靠前更性能底下。

基于ForEach循环(for(String str:list))、迭代器循环(for(Iterator iterator=list.iterator();iterator.hasNext()))，三者性能差不多，编译器编译代码时会将ForEach循环作为地带操作。但基于for循环(for(int i=0;i<list.size();i++))通过随机访问遍历列表时候，LinkedList性能很差，因为对LinkedList进行随机访问时，总会进行一次列表的遍历操作。

在读多写少的高并发环境中，使用CopyOnWriteArrayList可以提高系统的性能。但是，在读少写多的场合，CopyOnWriteArrayList性能不如Vector。

map接口

map接口的实现有：Hashtable、HashMap、LinkedHashMap和TreeMap。

1. HashMap/Hashtable

Hashtable与HashMap区别：

1. HashMap几乎可以等价于Hashtable，Hashtable大部分方法做了同步，HashMap没有，因此，HashMap不是线程安全的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。
2. Hashtable不允许key或者value使用null值，而HashMap可以。
3. 在内部算法上，他们对key的hash算法和hash值到内存索引的映射算法不同。

数据存储形式都一样，如下：

HashMap底层数据结构是数组，将value放到指定数组索引下有个算法：1. key.hashCode()获得数值，2. 使用hash(数值)取得一个hash值，3. 将hash值与数组长度length做&操作获取到数组索引，通过数组下标取得对应的值。

但数组索引肯定会冲突啊，即肯定会出现hash值相同情况，需要存放到HashMap中的两个元素1和2，通过hash计算后，发现对应在内存中的同一个地址。此时，HashMap又会如何处理以保证数据可以完整存放并正常工作呢？

HashMap底层实现使用数组，数组元素又使用Entry类的对象构成的链表。HashMap内部维护着一个Entry数组，每一个Entry表项包括key、value、hash值和next几项。其中next指向的是另外一个Entry。HashMap的put()方法，当put()操作有冲突时，新的Entry依然会被安放在对应的索引下标内，为保证旧值都不丢失，新的Entry的next指向旧值。所以HashMap实际上是一个链表的数组。

优化HashMap的性能：

如果key.hashCode()或者hash(数值)方法实现的足够好，尽量减少冲突的产生，HashMap的操作几乎等价于对数组的随机访问操作。单弱较差，在大量冲突发生的情况下，HashMap就退化为几个链表，对HashMap的操作等价于遍历链表，性能很差。

影响HashMap性能的还有它的容量参数。HashMap也是数组结构。HashMap构造函数提供了两个参数：初始容量和负载因子，初始容量即数组的大小，默认为16，扩容时，也一样，大小*2，负载因子又叫填充比，是介于0—1之间的浮点数，他决定了HashMap在扩容前其内部数组的填充度。默认为0.75。所以默认的HashMap扩容前的容量为 $16 \times 0.75 = 12$ 个。负载因子可以设置成>1的数，但若>1，必然会产生大量冲突，无疑是往10个口袋中放15个物品。所以当HashMap的实际大小超过threshold=初始容量*负载因子时，HashMap便会进行扩容。

2. LinkedHashMap

HashMap性能表现不错，但有个缺点是它的无序性，被存入到HashMap的元素，在遍历HashMap时，其输出是无序的。如果希望元素保存输入时的顺序，则需要使用LinkedHashMap替代。

LinkedHashMap继承自HashMap，它具备HashMap的高性能，同时，LinkedHashMap又在内部增加了一个链表，用以存放元素的顺序。因此，LinkedHashMap可以简单地理解为一个维护了元素次序表的HashMap。

LinkedHashMap可以提供：元素插入时的顺序和最近访问的顺序两种。构造方法包括3个参数：初始化容量、负载因子和排序类型，当排序类型为true时，按元素最近访问时间排序；为false时，按照插入顺序排序。默认为false。

LinkedHashMap通过继承HashMap.Entry类，实现了LinkedHashMap.Entry，为HashMap.Entry增加了before和after属性用以记录某一表项的前驱和后继Entry，每个Entry2表项的after指向其后继元素Entry，而Entry的before指向其前去元素Entry2，从而构成了循环链表。

3. TreeMap

TreeMap和HashMap完全不同，内部实现是基于红黑树，红黑树是一种平衡查找树，TreeMap又实现了SortedMap接口，他可以对元素进行排序，put进去的，默认出来就是按Key排序，若按value排序，可使用Comparable或Comparator两种方式，专门为排序而生，所以要对集合做排序，建议使用TreeMap。

5. ConcurrentHashMap

ConcurrentHashMap具体是怎么实现线程安全的呢，肯定不可能是每个方法加synchronized，那样就变成了HashTable。ConcurrentHashMap代码中引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。构造方法比HashMap多了一个N参数，根据segmentFor(hash(key.hashCode()))的值来决定把key放到哪个HashTable中。

在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中。

set接口

Set集合中的元素是不能重复的。主要有：HashSet、LinkedHashSet和TreeSet。所有对Set的实现，都只是对应的Map的一种封装而已。以HashSet为例，其内部维护一个HashMap对象，所有对Set的实现，都委托HashMap对象完成。

RandomAccess接口

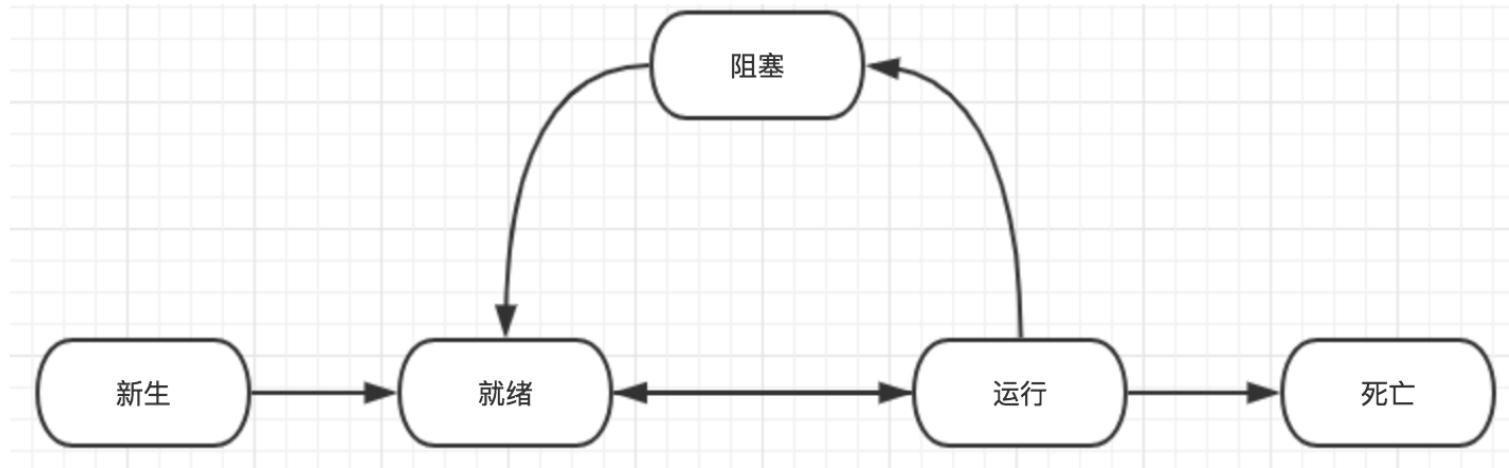
RandomAccess接口只是一个标志接口，本身没有提供任何方法，此接口的主要目标是标识那些可支持快速随机访问的List实现。任何一个基于数组的List实现都实现了RandomAccess接口，而基于链表的实现则都没有。if(list instanceof RandomAccess){for循环}else{ForEach循环/迭代循环}。

3 改善性能的技巧

1. 慎用异常
2. 局部变量的访问速度远远高于类的成员变量。调用方法时传递的参数以及调用中创建的局部变量都保存在栈中，速度较快。其他变量，如静态变量、实例变量等都在堆中创建，速度较慢。
3. 用位运算来代替乘法、除法以及取模
4. 对于数组复制，建议使用System.arraycopy()函数，native函数性能要优于普通的函数。

4 多线程

线程的生命周期



1. 新生状态

创建一个线程，未调用start()方法，该线程对象就处于新生状态。处于新生状态的线程有自己的内存空间，通过调用start方法进入就绪状态（runnable）。

注意：不能对已经启动的线程再次调用start()方法，否则会抛异常。

2. 就绪状态

处于就绪状态的线程已经具备了运行条件，但还没有分配到CPU，处于线程就绪队列，等待系统为其分配CPU。

3. 运行状态

处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。处于就绪状态的线程，如果获得了cpu的调度，就会从就绪状态变为运行状态。如果该线程失去了cpu资源，就会又从运行状态变为就绪状态。也可以对在运行状态的线程调用yield()方法，它就会让出cpu资源，再次变为就绪状态。

4. 阻塞状态

处于运行状态的线程在某些情况下，如执行了sleep（睡眠）方法，或等待I/O设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。

在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续运行。

5. 死亡状态

当线程的run()方法执行完，或者被强制性地终止，就认为它死去。线程一旦死亡，就不能复生。如果在一个死去的线程上调用start()方法，会抛异常。

同步关键字synchronized、wait、notify方法使用

当程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this实例）有关的锁。

一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获得锁，直到第一个线程释放（或返回）锁。这也意味着任何其他线程都不能进入该对象上的synchronized方法或代码块，直到该锁被释放。

要同步静态方法，需要一个用于整个类对象的锁，这个对象就是这个类 (XXX.class)。

例如：

```
1. public static synchronized int setName(String name){  
2.     Xxx.name = name;  
3. }
```

等价于

```
1. public static int setName(String name){  
2.     synchronized(Xxx.class){  
3.         Xxx.name = name;  
4.     }  
5. }
```

子线程循环10次，接着主线程循环100次，接着又回到子线程循环10次，接着主线程循环100次程序：

```
1. package com.itcase.thread;  
2. public class TraditionalThreadSynchronizedTest2 {  
3.     /**  
4.      * 让output1和output2互斥  
5.      *  
6.      * @param args  
7.      */  
8.     public static void main(String[] args) {  
9.         new TraditionalThreadSynchronizedTest2().init();  
10.    }  
11.    private void init() {  
12.        final Business business = new Business();  
13.        new Thread() {  
14.            public void run() {  
15.                for (int i = 1; i <= 50; i++) {  
16.                    business.sub(i);  
17.                }  
18.            };  
19.            }.start();  
20.            for (int i = 1; i <= 50; i++) {  
21.                business.main(i);  
22.            }  
23.        }  
24.        private class Business {  
25.            //不确定子线程和主线程哪个先执行，假如是先到主线程，主线程发现不是它执行，它就wait，这个时候子线  
26.            private boolean beShouldSub = true;  
27.            public synchronized void sub(int i) {  
28.                while(!beShouldSub){  
29.                    try {  
30.                        this.wait(); //不是它，它就等  
31.                    } catch (InterruptedException e) {  
32.                        e.printStackTrace();  
33.                    }  
34.                }  
35.                for (int j = 1; j <= 10; j++) {  
36.                    System.out.println("Sub thread sequence of " + i + " ,loop of "  
37.                                + j);  
38.                }  
39.                beShouldSub = false;  
40.                this.notify();  
41.            }
```

```
42.     public synchronized void main(int i) {
43.         while(beShouldSub){
44.             try {
45.                 this.wait(); //不是它， 它就等
46.             } catch (InterruptedException e) {
47.                 e.printStackTrace();
48.             }
49.         }
50.         for (int j = 1; j <= 100; j++) {
51.             System.out.println("main thread sequence of " + i
52.                               + " ,loop of " + j);
53.         }
54.         beShouldSub = true;
55.         this.notify();
56.     }
57. }
58. }
```

notify只能唤醒一个线程，其他等待的线程仍然处于wait状态，使用notifyAll()来唤醒所有正在等待该锁的线程。

interrupt()、stop()、join()、suspend()、resume()、yield()、守护线程

- `interrupt()`优雅停止线程，仅仅是在当前线程中打了一个停止的标记，并不是真正的停止线程。待线程内执行完后停止。

判断线程是否是停止状态，`Thread`类提供了两个方法：

2. `this.isInterrupted()` 测试线程是否已经中断。

- stop()暴露停止线程
 - join()让当前线程阻塞等待其他线程执行完毕，一般放在主线程里，等待子线程执行完毕。
 - suspend()暂停线程
 - resume()继续执行线程
 - yield()放弃当前CPU资源，让给其他的任务去占用CPU执行时间
 - 为线程.setDaemon(true);将线程设置成守护线程

ThreadLocal实现在当前线程范围任意地方调用都可以

```
1. public class ThreadLocalSharedData {
2.     public static ThreadLocal<Integer> threadData = new ThreadLocal<Integer>();
3.     public static void main(String[] args) {
4.         for (int i = 0; i < 2; i++) {
5.             new Thread(new Runnable() {
6.                 @Override
7.                 public void run() {
8.                     int data = new Random().nextInt(); // 随机一个整数
9.                     System.out.println(Thread.currentThread().getName()
10.                         + " has put data:" + data);
11.                     threadData.set(data);
12.                     new A().get();
13.                     new B().get();
14.                 }
15.             }).start();
16.         }
17.     }
18.     static class A {
19.         public void get() {
20.             int data = threadData.get();
21.             System.out.println("A from " + Thread.currentThread().getName()
22.                 + " has put data:" + data);
23.         }
24.     }
25. }
```

```
23. }
24. }
25. static class B {
26.     public void get() {
27.         int data = threadData.get();
28.         System.out.println("B from " + Thread.currentThread().getName()
29.                             + " has put data:" + data);
30.     }
31. }
32. }
```

volatile

在Java中，每一个线程有一块工作内存区，其中存放着被所有线程共享的主内存中的变量的值的拷贝。当线程执行时，它在自己的工作内存中操作这些变量。为了存取一个共享的变量，一个线程通常先获取锁定并且清除它的工作内存区，这保证该共享变量从所有线程的共享内存区正确地装入到线程的工作内存区，当线程解锁时保证该工作内存区中变量的值写回到共享内存中。

由于每个线程都有自己的工作内存区，因此当一个线程改变自己的工作内存中的数据时，对其他线程来说，可能是不可见的。为此，可以使用volatile关键字迫使所有线程都读写主内存中的对应变量，从而使得volatile变量在多线程间可见。

声明为volatile的变量可以做一下保证：

1. 其他线程对变量的修改，可以即时反应在当前线程中。
2. 确保当前线程对volatile变量的修改，能即时写回共享主内存中，并被其他线程所见。
3. 使用volatile声明的变量，编译器会保证其有序性。

CAS无锁实现原理

为什么要用CAS

在多线程高并发编程的时候，最关键的问题就是保证临界区的对象的安全访问。通常是用加锁来处理，其实加锁本质上是将并发转变为串行来实现的，势必会影响吞吐量。而且线程的数量是有限的，依赖于操作系统，而且线程的创建和销毁带来的性能损耗是不可以忽略掉的。虽然现在基本都是用线程池来尽可能的降低不断创建线程带来的性能损耗。

对于并发控制而言，锁是一种悲观策略，会阻塞线程执行。而无锁是一种乐观策略，它会假设对资源的访问时没有冲突的，既然没有冲突就不需要等待，线程不需要阻塞。那多个线程共同访问临界区的资源怎么办呢，无锁的策略采用一种比较交换技术CAS (compare and swap) 来鉴别线程冲突，一旦检测到冲突，就充实当前操作指导没有冲突为止。与锁相比，CAS使用无锁的方式没有锁竞争带来的开销，也没有线程间频繁调度带来的开销，他比基于锁的方式有更优越的性能，所以在目前被广泛应用，我们在程序设计时也可以适当的使用。

CAS算法

一个CAS方法包含三个参数CAS(V,E,N)。V表示要更新的变量，E表示预期的值，N表示新值。只有当V的值等于E时，才会将V的值修改为N。如果V的值不等于E，说明已经被其他线程修改了，当前线程可以放弃此操作，也可以再次尝试次操作直至修改成功。基于这样的算法，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰（临界区值的修改），并进行恰当的处理。

volatile实现了JMM中的可见性。使得对临界区资源的修改可以马上被其他线程看到，它是通过添加内存屏障实现的。

初次接触CAS的人一般都是通过AtomicInteger这个类来了解的，这里讲其CAS原理也借助这个类。

查看一下AtomicInteger的源码：

```
1. private volatile int value;
2. //此处省略一万字代码
3. /**
4.  * Atomically sets to the given value and returns the old value.
5. */
6. public final int getAndSet(int newValue) {
7.     for (;;) {
8.         int current = get();
9.         if (compareAndSet(current, newValue))
10.             return current;
11.     }
12. }
13. public final boolean compareAndSet(int expect, int update) {
14.     return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
15. }
```

通过这段代码可知：

- AtomicInteger中真正存储数据的是value变量，而改变量是被volatile修饰的，保证了线程直接的可见性。
- getAndSet方法通过一个死循环不断尝试赋值操作。而真正的赋值操作交给了unsafe类来实现。

Unsafe类是CAS实现的核心。

从名字可知，这个类标记为不安全的，JDK作者不希望用户使用这个类，所以它的构造方法如下：

```
1. public static Unsafe getUnsafe() {
2.     Class var0 = Reflection.getCallerClass();
3.     if(var0.getClassLoader() != null) {
4.         throw new SecurityException("Unsafe");
5.     } else {
6.         return theUnsafe;
7.     }
8. }
```

不允许哪里使用Unsafe。由于CAS编码稍微复杂，而且jdk作者本身也不希望你直接使用unsafe来进行代码的编写，所以如果不能深刻理解CAS以及unsafe还是要慎用，使用一些别人已经实现好的无锁类或者框架就好了。

看看[unsafe.compareAndSwapInt\(this, valueOffset, expect, update\);](#)方法

参数1 当前对象

参数2 offset是对象内的偏移量（其实就是一个字段到对象头部的偏移量，通过这个偏移量可以快速定位字段）

参数3 期望值

参数4 要设置的值。

这里Unsafe封装了一些类似于C++中指针的东西，该类中的方法都是native的，而且是原子的操作。原子性是通过CAS原子指令实现的，由处理器保证。

java.util.concurrent.atomic下所有包都采用CAS来实现无锁。

java.util.concurrent包

java.util.concurrent.atomic包

AtomicBoolean、AtomicInteger、AtomicLong等，提供了原子增量方法。

```
1. class Counter {
2.     private AtomicInteger count = new AtomicInteger();
3.     public void increment() {
4.         count.incrementAndGet();
```

```
5. }  
6. // 使用AtomicInteger之后，不需要加锁，也可以实现线程安全。  
7. public int getCount() {  
8.     return count.get();  
9. }  
10. }
```

java.util.concurrent.locks

提供Lock锁和Qt的QMutex一样、ReadWriteLock读写锁。Condition、Semaphore信号量

互斥锁Lock是一个接口。提供了无条件的、可轮询的、定时的、可中断的锁获取操作。ReentrantLock(重入锁)是Lock的实现。

读写锁ReadWriteLock，维护了一对相关的锁，一个用于只读操作，一个用于写入操作。

ReentrantReadWriteLock是ReadWriteLock的实现。

互斥锁一次只允许一个线程访问共享数据，哪怕进行的是只读操作；读写锁允许对共享数据进行更高级别的并发访问：对于写操作，一次只有一个线程（write线程）可以修改共享数据，对于读操作，允许任意数量的线程同时进行读取。

Lock与synchronized 的比较：

- 1: Lock使用起来比较灵活，但是必须有释放锁的动作；
- 2: Lock必须手动释放和开启锁，synchronized 不需要；
- 3: Lock只适用于代码块锁，而synchronized 对象之间的互斥关系；

公平锁：锁的获取是有序的，等待时间最长的线程最有机会获取锁。否则是不公平锁。

ReentrantLock通过在构造方法中可指定是否公平，不指定默认为非公平锁。公平锁是通过一个等待队列来实现的公平锁。

ReentrantReadWriteLock也是基于AbstractQueuedSynchronizer等待队列实现的公平锁和不公平锁，和ReentrantLock一样。

JDK1.5之后的并发包中提供的CountDownLatch也可以实现join的功能，并且比join的功能更多。

CountDownLatch的构造函数接收一个int类型的参数作为计数器，如果你想等待N个点完成，这里就传入N。当我们调用一次CountDownLatch的countDown方法时，N就会减1，CountDownLatch的await会阻塞当前线程，直到N变成零。由于countDown方法可以用在任何地方，所以这里说的N个点，可以是N个线程，也可以是1个线程里的N个执行步骤。用在多个线程时，你只需要把这个CountDownLatch的引用传递到线程里。

BlockingQueue一个指定长度的队列，如果队列满了，添加新元素的操作会被阻塞等待，直到有空位为止。同样，当队列为空时候，请求队列元素的操作同样会阻塞等待，直到有可用元素为止。DelayQueue，时间到了才出队列。

LinkedBlockingDeque栈是“后入先出”的结构，每次操作的是栈顶，而队列是“先进先出”的结构，每次操作的是队列头。

java.util.concurrent.Executors

Java5提供5种类型的线程池，分别如下：

- 1.newCachedThreadPool-可变尺寸的线程池(缓存线程池)
- 2.newFixedThreadPool-固定大小的线程池

3.ScheduledThreadPool-调度线程池 这个池子里的线程可以按schedule依次delay执行，或周期执行。

4.SingleThreadExecutor-单例线程池 任意时间池中只能有一个线程；

5.自定义线程池--ThreadPoolExecutor

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。

```
1. public class CallableFutureTest {
2.
3.     class MyCallable implements Callable {
4.         private String name;
5.         MyCallable(String name) {
6.             this.name = name;
7.         }
8.         public Object call() throws Exception {
9.             return name + "任务返回的内容";
10.        }
11.    }
12.
13.    @SuppressWarnings("unchecked")
14.    public static void main(String[] args) throws ExecutionException,
15.        InterruptedException {
16.        CallableFutureTest test = new CallableFutureTest();
17.        // 创建一个线程池
18.        ExecutorService pool = Executors.newFixedThreadPool(2);
19.        // 创建两个有返回值的任务
20.        Callable c1 = test.new MyCallable("A");
21.        Callable c2 = test.new MyCallable("B");
22.        // 执行任务并获取Future对象
23.        Future f1 = pool.submit(c1);
24.        Future f2 = pool.submit(c2);
25.        // 从Future对象上获取任务的返回值，并输出到控制台
26.        System.out.println(">>>" + f1.get().toString());
27.        System.out.println(">>>" + f2.get().toString());
28.        // 关闭线程池
29.        pool.shutdown();
30.    }
31.}
```

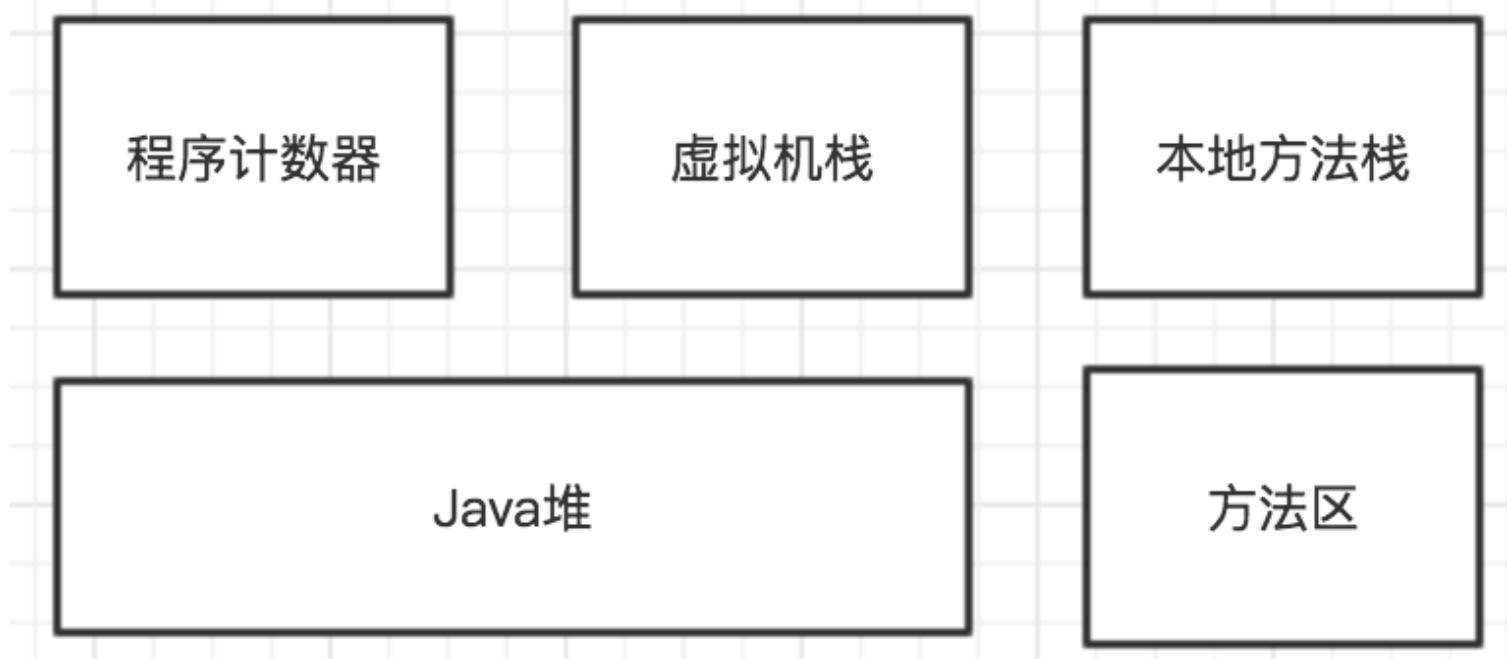
```
1. public class ThreadPoolTest {
2.     public static void main(String[] args) {
3.         ExecutorService threadPool = Executors.newFixedThreadPool(3);
4.         for(int i = 1; i < 5; i++) {
5.             final int taskID = i;
6.             threadPool.execute(new Runnable() {
7.                 public void run() {
8.                     for(int i = 1; i < 5; i++) {
9.                         try {
10.                             Thread.sleep(20); // 为了测试出效果，让每次任务执行都需要一定时间
11.                         } catch (InterruptedException e) {
12.                             e.printStackTrace();
13.                         }
14.                         System.out.println("第" + taskID + "次任务的第" + i + "次执行");
15.                     }
16.                 }
17.             });
18.         }
19.         threadPool.shutdown(); // 任务执行完毕，关闭线程池
20.     }
21. }
```

Amino无锁算法框架、Kilim协程。

5 JVM调优

Java虚拟机内存模型

JVM虚拟机将其内存数据分为程序计数器、虚拟机栈、本地方法栈、Java堆和方法区等部分。



- 程序计数器用于存放下一条运行的指令；
- 虚拟机栈和本地方法栈用于存放函数调用堆栈信息；
- Java堆用于存放Java程序运行时所需的对象等数据；
- 方法区用于存放程序的类元数据信息。

程序计数器

每一个线程都有一个独立的程序计数器，当正在执行一个java方法，程序计数器记录的是正在执行的Java字节码地址；如果正在执行的是一个Native方法，则程序计数器为空；

Java虚拟机栈

Java虚拟机栈也是线程私有的内存空间，它和Java线程在同一时间创建，它保存方法的局部变量、部分结果，并参与方法的调用和返回。

在Hot Spot虚拟机中，可以使用-Xss参数来设置栈的大小。栈的大小直接决定了函数调用的可达深度。

如递归无限制的调一个方法做测试，可得出结论：可递归的深度和栈大小是决定的。栈越大，函数嵌套调用次数越多。

虚拟机栈在运行时使用一种叫做栈帧的数据结构保存上下文数据。在栈帧中，存放了方法的局部变量表（包括方法的参数和方法内的局部变量）、操作数栈、动态连接方法和返回地址等信息。每一个方法的调用都伴随着栈帧的入栈操作。相应地，方法的返回则表示栈帧的出栈操作。对于一个函数而言，它的参数越多，内部局部变量越多，它的栈帧就越大，其嵌套调用次数就会减少。局部变量表以“字”为单位进行内存划分，一个字为32位长度。对于long,double变量占2个字，其他类型占1个字。如果请求的栈深度大于最大可用的栈深度，抛出StackOverflowError；没有足够的内存空间来支持栈的扩展，则抛出OutOfMemoryError。

本地方法栈

本地方法栈是用于管理本地方法(C实现的)的调用，在SUN的Hot Spot虚拟机中，不区分本地方法栈和虚拟机栈。和虚拟机栈一样，它也会跑出StackOverflowError和OutOfMemoryError。

Java堆

Java堆，几乎所有对象和数组都是在堆中分配空间的，所有线程共享的。Java堆分为新生代和老年代两部分。新生代用于存放刚刚产生的对象和年轻的对象，如果对象一直没有被回收，生存的足够长，老年对象就会被移入老年代。

新生代又可细分为eden(出生地，大部分对象刚刚建立时，通常会存放在那里)、survivor space0(so或者from space)和survivor space1(s1或者to space)，s0和s1为survivor空间，译为幸存者，也就是至少经过一次垃圾回收后还得以幸存的。如果在幸存区的对象到了指定年龄仍未被回收，则有机会进入老年代(tenured)。

后面讲JVM GC算法时详细讲。

方法区

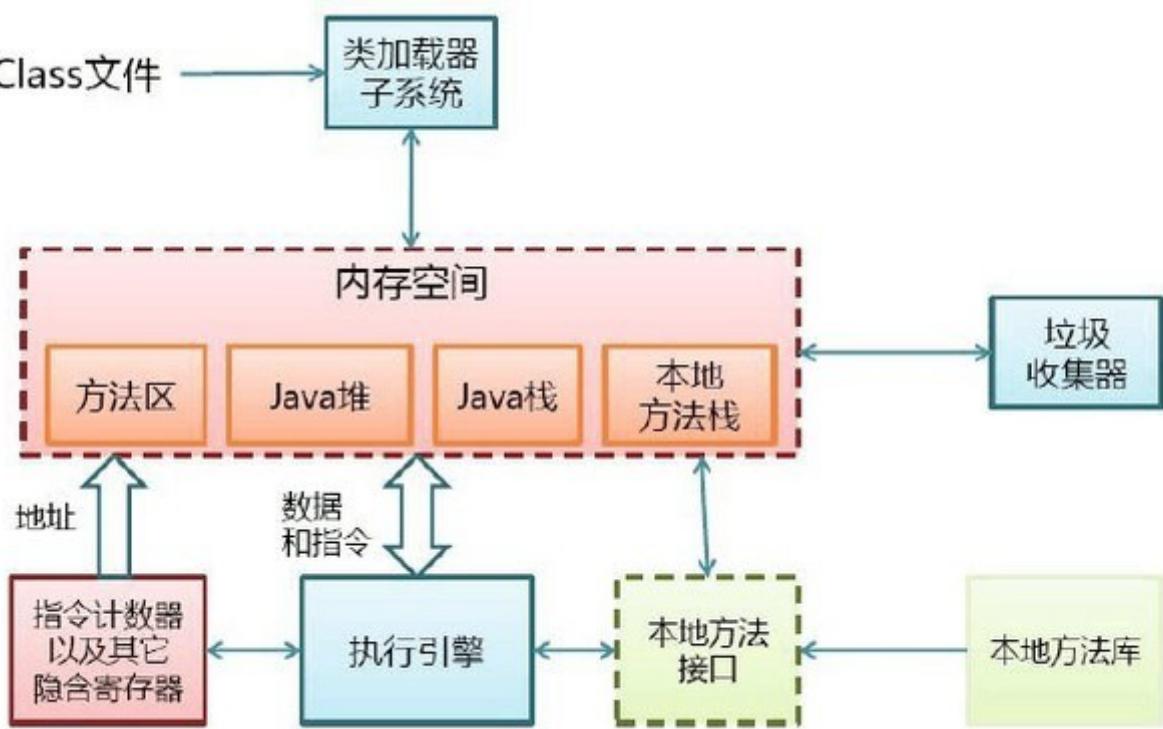
也是被JVM所有线程共享的。方法区主要保存的信息是类的元数据，包括类的完整信息（完整名称、父类信息，类型修饰符、类的接口表等）、常量池、方法信息（方法名、返回类型、方法参数等）。总之，方法区内保存的信息，大部分来自于class文件，是Java应用程序必不可少的重要数据。

在Hot Spot虚拟机中，方法区也称为永久区，是一块独立的内存空间。但GC也会回收永久区的，回收常量池、回收类元数据。如果常量池中的常量没有被任何地方引用，就可以被回收。

在现有的软件开发项目中，CGLIB和Javassist等动态字节码生成工具得到非常普遍的使用。当系统中需要生成大量动态类时，对持久化的压力会非常大。持久代若饱和，会抛出“java.lang.OutOfMemoryError:PermGen space”显示持久代溢出。

JVM原理

1. 每个Java虚拟机都由一个类加载器子系统负责加载程序中的类型（类和接口），并赋予唯一的名字，都有一个执行引擎负责执行被加载类中包含的指令。
2. 每个Java虚拟机都包含方法区和堆，他们都被整个程序共享。Java虚拟机加载并解析一个类以后，将从类文件中解析出来的信息保存于方法区中。程序执行时创建的对象都保存在堆中。当一个线程被创建时，会被分配只属于他自己的PC寄存器和Java堆栈。当线程不调用本地方法时，PC寄存器中保存线程执行的下一条指令。Java堆栈保存了一个线程调用方法时的状态，包括本地变量、调用方法的参数、返回值、处理的中间变量。调用本地方法时的状态保存在本地方法堆栈中。
3. 当Java程序创建一个类的实例或者数组时，都在堆中为新的对象分配内存。虚拟机中只有一个堆，所有的线程都共享他。Java堆栈由堆栈块组成。堆栈块包含Java方法调用的状态。当一个线程调用一个方法时，Java虚拟机会将一个新的块压到Java堆栈中，当这个方法运行结束时，Java虚拟机会将对应的块弹出并抛弃。
4. Java虚拟机不使用寄存器保存计算的中间结果，而是用Java堆栈在存放中间结果。



此图看出jvm内存结构

JVM内存结构主要包括两个子系统和两个组件。两个子系统分别是Classloader子系统和Executionengine(执行引擎)子系统；两个组件分别是Runtimedataarea(运行时数据区域)组件和Nativeinterface(本地接口)组件。

- Classloader子系统的作用：根据给定的全限定名类名(如java.lang.Object)（就是带包的类路径）来装载class文件的内容到JVM内存（Runtimedataarea）中的methodarea(方法区域)。
- Executionengine子系统的作用：执行classes中的指令。
- Nativeinterface组件：是其它编程语言交互的接口。当调用native方法的时候，就进入了一个全新的并且不再受虚拟机限制的世界，所以也很容易出现JVM无法控制的nativeheapOutOfMemory。
- RuntimeDataArea组件：这就是我们常说的JVM的内存了。它主要分为五个部分：
 - Heap(堆)：一个Java虚拟实例中只存在一个堆空间
 - MethodArea(方法区域)：被装载的class的信息存储在Methodarea的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的class文件，然后读入这个class文件内容并把它传输到虚拟机中。
 - Java Stack(java的栈)：虚拟机只会直接对Javastack执行两种操作：以帧为单位的压栈或出栈
 - Program Counter(程序计数器)：每一个线程都有它自己的PC寄存器，也是该线程启动时创建的。PC寄存器的内容总是指向一条将被执行指令的地址
 - Native method stack(本地方法栈)：保存native方法进入区域的地址。

常见参数设置

- 最大堆大小：-Xmx 最大堆指的是新生代和老年代的大小和的最大值。
- 最小堆内存：-Xms 启动JVM时，所占据的操作系统的内存大小。Java应用程序运行时，首先会被分配-Xms大小内存，尽可能在该内存下运行，当内存大小确实无法满足应用程序运行时，JVM才会向操作系统申请更多的内存。直到-Xms，若还无法满足，则抛出OutOfMemoryError异常。如果-Xms值较小，JVM为了保证系统尽可能在指定范围内运行，就会更加频繁地进行GC操作，以释放失效的内存空间。因此把-Xms值设置为-Xmx时，可以在系统运行初期减少GC的次数和耗时。
- 设置新生代：-Xmn 设置较大的新生代会减少老年代的大小，新生代大小一般设置为整个堆空间的1/4到1/3左右。

在Hot Spot虚拟机中，同样支持-XX:NewSize设置新生代的初始大小，-XX:MaxNewSize设置新生代的最大值，一般不用设置。

- 设置持久代 持久代不属于堆的一部分，就是方法区。使用-XX:MaxPermSize可以设置持久代的最大值，使用-XX:PermSize可以设置持久代的初始大小。持久代的大小直接决定了系统可以支持多少个类定义和多少常量。对于使用CGLIB或者Javassist等动态字节码生成工具的应用程序，应合理设置持久代大小。
- 设置线程栈大小 :-Xss 函数调用时，都需要在栈中开辟空间，如果栈分配空间太小，会影响调用函数深度；同时，如果栈空间过大，所能支持的线程总数就会下降。这两个是矛盾，但要均衡。
- 对堆里设置比例：-XX:SurvivorRatio可以设置eden区与survivor区的比例。-XX:NewRatio可以设置老年代与新生代的比例。

class文件结构

class文件结构：4个字节class文件标识符，jdk小版本，大版本，常量数，常量表，访问标识符，类名，父类名，实现接口数，实现接口表，字段数，字段表，方法数，方法表，属性数，属性表。每个表都放了详细信息，如访问控制符，synchronized等。在加载字节码class文件时候，会用到他们。

垃圾收集器

一个不合适的垃圾回收方法和策略会对系统性能造成不良影响。本节介绍一些垃圾回收方法以及Hot Spot虚拟机支持的垃圾回收器。

常用垃圾收集算法

- 引用计数法

对于每个对象配备一个计数器，记录哪里有引用了它，当有引用，计数器+1，当引用时效，计数器-1，只要对象的引用计数器的值为0，则对象就不可能再被使用。但引用计数器无法处理循环引用问题，A引用B，B引用A，所以不合适做Java垃圾回收。

- 标记清除算法

标记-清除算法先通过根节点标记所有可达对象，然后清除所有不可达对象，这样回收后的空间是不连续的。在对象的堆空间分配过程中，尤其是大对象的内存分配，不连续的内存空间的工作效率要低于连续的空间。

- 复制算法

将内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中（连续放），然后清除这个内存块，完成垃圾回收。复制算法效率很高，这是在存活对象少，垃圾对象多情况，而且复制算法将系统内存折半，单纯的复制算法让人不能接受。

在Java的新生代串行垃圾回收器中，使用了复制算法的思想。新生代分为eden空间、from空间和to空间3个部分。其中from和to空间可以视为用于复制的两块大小相同、地位相等，且可进行角色互换的空间块。from和to空间也称为survivor空间，即幸存者空间，用于存放未被回收的对象。在垃圾回收时，eden空间的存活对象会被复制到未使用的survivor空间中（假设是to），正在使用的survivor空间（假设是从）中的年轻对象也会被复制到to空间中（大对象，或者老年对象会直接进行年老代，如果to空间已满，则对象也会直接进行老年代）。此时，eden空间和from空间中的剩余对象就是垃圾对象，可以直接清空，to空间则存放此时回收后的存活对象。

但对于Java老年代，存活对象过多，这种方法效率就不高了。

- 标记—压缩算法

标记—压缩算法是一种老年代的回收算法，它对标记-清除算法做了优化。它也是从根节点开始，对所有可达对象做一次标记。标记完后，它将所有存活对象压缩到内存的一端。之后，清理边界外所有的空间（都放到一端，然后清理这一端意外的所有）。性价比较高。

- 分代

分代是同时使用赋值算法和标记-压缩算法，它将内存区间根据对象特点分成2大块，年轻代的特点是对象朝生夕灭，大约90%的新建对象会被很快回收，因此，年轻代使用效率较高的复制算法。当一个对象经过几次回收后依

然存活，对象就会被放入称为老年代的内存空间。在老年代中，几乎所有对象都是经过几次垃圾回收后依然得以幸存的。可认为，这些对象是常驻内存的。对老年代使用标记-压缩算法。

对于任何的垃圾回收过程中，所有的线程都会被挂起，暂停一切正常的工作，等待垃圾回收的完成。如果垃圾回收时间很长，应用程序就会被挂起很久，将严重影响用户体验或者系统的稳定性。

垃圾收集器类型：

在执行机制上JVM提供了串行GC(SerialGC)、并行GC(ParallelGC)和并发GC(CMS)? Par是什么

1)串行GC

使用的是标记-压缩算法，在整个扫描和复制过程采用单线程的方式来进行，适用于单CPU、新生代空间较小及对暂停时间要求不是非常高的应用上，是client级别默认的GC方式，可以通过-XX:+UseSerialGC来强制指定

2)并行GC

使用的是标记-压缩算法，JVM的默认垃圾回收器，只是把串行GC变成多核GC，在整个扫描和复制过程采用多线程的方式来进行，适用于多CPU、对暂停时间要求较短的应用上，是server级别默认采用的GC方式，可用-XX:+UseParallelGC来强制指定，用-XX:ParallelGCThreads=4来指定线程数。

3) 并发GC

CMS收集器关注于系统停顿时间。使用的是标记-清除算法，同时也是使用的多线程并行回收的垃圾收集器。不是独占式回收器，在回收过程中，应用程序仍然再不停地工作。但因为是标记-清除算法会造成大量内存碎片，离散的可用空间无法分配较大的对象，即使堆内存仍然有较大的剩余空间，也可能会被迫进行一次垃圾回收。

常用调优案例和方法

老生代进行一次垃圾清理，被称为fullGC或者majorGC

新生代进行一次垃圾清理，被称为youngGC或者minorGC

我们在JVM优化过程中的一个原则就是：

降低youngGC的频率、减少fullGC的次数。

我们需要根据应用场景、硬件性能和吞吐量需求来决定使用哪一种垃圾回收器，通过JVM参数的配置来选择垃圾回收器：

1. -XX:+UseSerialGC: 串行垃圾回收器
2. -XX:+UseParallelGC: 并行垃圾回收器
3. -XX:+UseConcMarkSweepGC: 并发标记扫描垃圾回收器
4. -XX:ParallelCMSThreads: 并发标记扫描垃圾回收器 =为使用的线程数量
5. -XX:+UseG1GC: G1垃圾回收器

其他内存常见配置：

1. -Xms: 初始化堆内存大小
2. -Xmx: 堆内存最大值
3. -Xmn: 新生代大小
4. -XX:PermSize: 初始化永久代大小
5. -XX:MaxPermSize: 永久代最大容量

配置JVM GC参数的实例：

1. java -Xmx1024m -Xms512m -Xmn256m -XX:PermSize=64m -XX:MaxPermSize=128m -XX:+UseSerialGC

上述配置的含义是：

堆内存512-1024M，新生代256M，永年代64-128M，采用串行垃圾回收器执行java-application.jar。

java.lang.OutOfMemoryError: PermGen space Perm区内存溢出

java.lang.OutOfMemoryError: heap space 堆区内存溢出

JVM调优的主要过程有：确定堆内存大小（-Xms、-Xmx）、合理分配新生代和老年代（-XX:NewRatio、-Xmn、-XX:SurvivorRatio）、确定永久区大小（-XX:PermSize、-XX:MaxPermSize）、选择垃圾收集器/对垃圾收集器进行合理的设置，如设置线程数等。除此之外，禁用显示GC（-XX:+DisableExplicitGC，禁止再代码中System.gc()出发Full gc），禁用类元数据回收（-Xnoclassgc），禁用类验证（-Xverify:none）等设置，对提升系统性能也有一定的帮助。

常用调优工具

- top 查看各个进程的资源占用状况
- pidstat 既可以监视进程性能情况，也可以监视线程的性能情况。

-p 监控CPU使用率

-d 监控磁盘IO

-r 监控内存使用情况

如对CPU使用率做如下分析

查看进程CPU使用率

```
1. #每秒采样1次，采样3次，采CPU使用率  
2. pidstat -p 进程ID -u 1 3
```

```
1. #每秒采样1次，采样3次，采CPU使用率 -t可以将监控细化到线程级别  
2. pidstat -p 进程ID -u -t 1 3
```

```
1. jstack -l 进程ID > 1.txt
```

jstack导出Java应用程序的线程堆栈，导出信息到文件中

```
1. "Timer-0" #25 daemon prio=5 os_prio=31 tid=0x00007fb9b748c000 nid=0x11353 in Object.wait()  
2.     java.lang.Thread.State: WAITING (on object monitor)  
3.         at java.lang.Object.wait(Native Method)  
4.         at java.lang.Object.wait(Object.java:502)  
5.         at java.util.TimerThread.mainLoop(Timer.java:526)  
6.             - locked <0x0000007a1597500> (a java.util.TaskQueue)  
7.         at java.util.TimerThread.run(Timer.java:505)  
8.  
9.     Locked ownable synchronizers:  
10.        - None  
11. ...
```

这里的nid值刚好就是进程ID。所以根据这个nid可以确定在Java应用程序中大量占用CPU的线程。

- jps 查看所有Java程序的进程ID以及Main函数、程序启动参数，主函数完整路径等信息
- jstat 查看Java应用程序的堆使用情况以及GC情况。

```
1. jstat -options 进程ID
```

可以列出当前JVM版本支持的选项，常见的有

[options选项]

class 加载类的信息，数量，所占空间
compiler 显示编译相关信息，编译数、失败数
gc 显示eden、s0、s1、old使用量，总量。Yong gc次数，Full gc次数，最后一次耗时
gccapacity 和上面类似
gcutil 显示eden、s0、s1、old使用率。Yong gc次数，Full gc次数，最后一次耗时
...
• visual VM是一个可代替jstat、jmap、jhat、jstack、JConsole的工具。
• jstack 导出Java应用程序的线程堆栈。

Java精粹代码

定义方法时，使用...代表可以使用任意个参数

```
1. public class ReflactTest {  
2.     /**  
3.      * @param args  
4.     */  
5.     public static void main(String[] args) {  
6.         new ReflactTest().test(1);  
7.         new ReflactTest().test(1, 2);  
8.         new ReflactTest().test(1, 2, 2, 42);  
9.     }  
10.    public void test(int... yp) {  
11.        for (int i = 0; i < yp.length; i++) {  
12.            System.out.println(yp[i]);  
13.        }  
14.    }  
15. }
```

泛型中使用?、super、extends

```
1. public class Test {  
2.     /**  
3.      * @param args  
4.     */  
5.     public static void main(String[] args) {  
6.         Map<Integer, Integer> map = new HashMap<Integer, Integer>();  
7.         map.put(1, 1);  
8.         List<String> list = new ArrayList<String>();  
9.         list.add("123");  
10.        //限定通配符的上边界。?任意类型。Number包括八大基本类型，Number或者Number的8大子类  
11.        Vector<? extends Number> ver = new Vector<Integer>();  
12.        // Vector<? extends Number> ver2 = new Vector<String>();//错误。类型必须是Number或者Num  
13.        //限定通配符的下边界。?任意类型。Integer或者Integer的父类都可以  
14.        Vector<? super Integer> sub = new Vector<Number>();  
15.        // Vector<? super Number> sub2 = new Vector<String>();//错误。类型必须是Integer或者Inte  
16.        // Vector<? super Number> sub3 = new Vector<boolean>();//错误。类型必须是Integer或者Inte  
17.    }  
18.    public static void printf(Collection<?> collecation){//?代表接收任何类型的集合，如果要表示任  
19.        if(collecation!=null && collecation.size()>0){  
20.            if(collecation instanceof Arrays){  
21.                for(Object col : collecation){
```

```

22.         if(col instanceof String){
23.             System.out.println(col);
24.         }else if(col instanceof MyClass){
25.             System.out.println(col.toString());
26.         }
27.     }
28. }else{
29.     throw new RuntimeException("必须为set/list");
30. }
31. }
32. }
33. //限定通配符的上边界。?任意类型。Number包括八大基本类型，Number或者Number的8大子类
34. public List<? super Number> getList(){
35.     return null;
36. }
37. //限定通配符的下边界。?任意类型。Integer或者Integer的父类都可以
38. public List<? extends Number> getList2(){
39.     return null;
40. }
41. public class MyClass{
42.     @Override
43.     public String toString() {
44.         return "test";
45.     }
46. }
47. }

```

break、continue语句结合语句标签的使用

```

1. public class TestLabel {
2.     public static void main(String[] args) {
3.         outer:
4.         for (int i = 0; i < 10; i++) {
5.             System.out.println("\nouter_loop:" + i);
6.             inner:
7.             for (int k = 0; i < 10; k++) {
8.                 System.out.print(k + " ");
9.                 int x = new Random().nextInt(10);
10.                if (x > 7) {
11.                    System.out.print(" >>x == " + x + ", 结束inner循环,
12.                    continue outer;
13.                }
14.                if (x == 1) {
15.                    System.out.print(" >>x == 1, 跳出并结束整个outer和in
16.                    break outer;
17.                }
18.            }
19.        }
20.        System.out.println("----->>所有循环执行完毕! ");
21.    }
22. }

```

抽象接口类、匿名类的使用

```

1. public abstract class BaseSender {

```

```

1. /**
2.  * 发送 必须让子类实现才能使用
3. */
4. public abstract String send(String to);
5. public String getContent(String to) {
6.     this.send(to);
7.     System.out.println("BaseSender getContent");
8.     return null;
9. }
10.
11. }

13. public class SenderTest {
14.     public static void main(String[] args) {
15.         Sender sender3 = new BaseSender() {// 匿名类。
16.             @Override
17.             public String send(String senderID, String senderNumber,
18.                 String receiverID, String receoverNumber) {
19.                 return null;
20.             }
21.         };
22.         sender3.getContent("123");// 引用过时的方法
23.     }
24. }

```

使用回调模式

```

1. public interface Person {
2.     public int getAge();
3.     public String getName();
4. }
5. public class LynzaboUtil {
6.     public String sayHello(Person person) {
7.         return person.getName() + ",你好啊!";
8.     }
9. }
10. public class MainTest {
11.     /**
12.      * @param args
13.     */
14.     public static void main(String[] args) {
15.         LynzaboUtil util = new LynzaboUtil();
16.         final String operator = "习近平";
17.         String info = util.sayHello(new Person() {// 重写接口或者接口的抽象类都可以。
18.             @Override
19.             public int getAge() {
20.                 return 10;
21.             }
22.             @Override
23.             public String getName() {
24.                 return operator;
25.             }
26.         });
27.         System.out.println(info);// 习近平,你好啊!
28.     }
29. }

```

使用lambda

```
1. // Java 8之前:
```

```

2.     new Thread(new Runnable() {
3.         @Override
4.         public void run() {
5.             System.out.println("Before Java8, too much code for too little to do");
6.         }
7.     }).start();
8.     1
9.     2
10.    //Java 8方式:
11.    new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!"))

```

数字字面量下划线支持

(当遇到很长的数字，我们采取的是分段分隔的方式。比如数字500000，我们通常会写成500,000，即每三位数字用逗号分隔。利用这种方式就可以很快知道数值的大小。这种做法的理念被加入到了Java 7中，不过用的不是逗号，而是下划线“_”，System.out.println(1_500_000); //输出1500000)

jdk1.8 ([TODO 好难，最多看看lambda，还有函数式接口](#))

允许在接口中有默认方法实现

```

1.  public interface Animal {
2.      void sayHi();
3.      default void eat() {
4.          System.out.println("animal eat default method");
5.      }
6.  }
7.  public class Dog implements Animal {
8.      public void sayHi() {
9.          System.out.println("dog");
10.     }
11.     public static void main(String args[]) {
12.         Dog dog = new Dog();
13.         dog.eat(); // 再具体的类里面不是必须重写默认方法，但必须要实现抽象方法。
14.     }
15. }
16. }

```

Java版本对比

jdk新特性总结：

jdk1.5

枚举、静态导入、函数可变参数、提供一套内省方法，可以直接访问某个属性的getter/setter方法、支持泛型、For-Each循环(for(String a:list)类型)、注解。

jdk1.6

将复杂易变逻辑放入js中，支持运行Javascript、内置轻量级Http Server、内置嵌入式数据库Derby。

jdk1.7

1. switch中可以使用字符串

2. 运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断

3. 数字字面量下划线支持 (当遇到很长的数字，我们采取的是分段分隔的方式。比如数字500000，我们通常会写成500,000，即每三位数字用逗号分隔。利用这种方式就可以很快知道数值的大小。这种做法的理念被加入到了Java 7中，不过用的不是逗号，而是下划线“_”，`System.out.println(1_500_000); //输出1500000`)

jdk1.8 ([TODO 好难，最多看看lambda，还有函数式接口](#))

1. 允许在接口中有默认方法实现

```
1. public interface Animal {
2.     void sayHi();
3.     default void eat() {
4.         System.out.println("animal eat default method");
5.     }
6. }
7. public class Dog implements Animal {
8.     public void sayHi() {
9.         System.out.println("dog");
10.    }
11.
12.    public static void main(String args[]) {
13.        Dog dog = new Dog();
14.        dog.eat(); // 再具体的类里面不是必须重写默认方法，但必须要实现抽象方法。
15.    }
16. }
```

2. Lambda表达式

3. 函数式接口

4. 方法和构造函数引用

5. Lambda的范围

6. 内置函数式接口

7. Streams

8. Parallel Streams

10. 时间日期API

11. Annotations

Spring、SpringMVC、Spring Cloud

Spring IOC

Spring IOC控制反转 负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

IoC 控制反转，指将对象的创建权，反转到Spring容器， DI 依赖注入，指Spring创建对象的过程中，将对象依赖属性通过配置进行注入。

在Java中依赖注入有以下三种实现方式：

构造器注入

Setter方法注入

接口注入

Spring框架支持如下五种不同的作用域：

- singleton: 在Spring IOC容器中仅存在一个Bean实例，Bean以单实例的方式存在。
- prototype: 一个bean可以定义多个实例。
- request: 每次HTTP请求都会创建一个新的Bean。该作用域仅适用于WebApplicationContext环境。
- session: 一个HTTP Session定义一个Bean。该作用域仅适用于WebApplicationContext环境。
- globalSession: 同一个全局HTTP Session定义一个Bean。该作用域同样仅适用于WebApplicationContext环境.全局作用域与Servlet中的session作用域效果相同。

Spring框架中单例beans是线程安全的吗?

不是，Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。

Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Spring生命周期：

在一个bean实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个bean不在被调用时需要进行相关的析构操作，并从bean容器中移除。

Spring bean在初始化时加载，你可以在bean上加lazy-init="true"，则这个bean在第一次使用时候创建。

Spring自动装配提供很多模式：主要是byName、byType。使用自动注解，可以消除相应的set和get方法。

@Autowired注解与@Resource注解的区别：

@Autowired只按照byType 注入； @Resource默认按byName自动注入，也提供按照byType 注入；
在二者上可以加：

@Required注解用于检查特定的属性是否必须存在，可以设置不是必须注入。

@Qualifier明确指定注入的是哪个。比如beans定义了id不同的多个同一个bean， id="student1"和id="student2"，当使用@Autowired按类型时，可以再加

```
1. @Autowired
2. @Qualifier("student2") //明确指定使用第二个
3. private Student student;
```

Spring AOP

面向切面编程

使用JDK中的Proxy技术和CGLIB库文件实现AOP功能。

AOP的核心就是切面，它将多个类的通用行为封装为可重用的模块。

连接点(Join point) 代表应用程序中插入AOP切面的地点。

通知(Advice) 表示在方法执行前后需要执行的动作。

Spring切面可以执行一下五种类型的通知：

- before(前置通知): 在一个方法之前执行的通知。
- after(最终通知): 当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
- after-returning(后置通知): 在某连接点正常完成后执行的通知。
- after-throwing(异常通知): 在方法抛出异常退出时执行的通知。

- around(环绕通知): 在方法调用前后触发的通知。

切入点(Pointcut)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。

目标对象是被一个或者多个切面所通知的对象。

织入是将切面和其他应用类型或对象连接起来创建一个通知对象的过程。织入可以在编译、加载或运行时完成。

代理是将通知应用到目标对象后创建的对象。

Spring提供了两种切面声明方式，实际工作中我们可以选用其中一种：

- 基于XML配置方式（基于aop:config和aop:aspect）声明切面。
- 基于注解方式(基于@AspectJ)声明切面

基于XML Schema方式的切面实现，切面由使用XML文件配置的类实现。

基于注解方式(基于@AspectJ)的切面实现，指的是切面的对应的类使用Java 5注解的声明方式。

AOP事务管理

Spring的事务管理分为注解方式（直接给类或方法加@Transactional）与XML配置文件（用XML配置来管理事务）。

XML配置：

```

1. <!-- 使用该配置可以读取properties文件 -->
2. <context:property-placeholder location="jdbc.properties" />
3. <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
4.     destroy-method="close">
5.     <property name="driverClassName" value="${driverClassName}" />
6.     <property name="url" value="${url}" />
7.     <property name="username" value="${username}" />
8.     <property name="password" value="${password}" />
9.     <!-- 连接池启动时的初始值 -->
10.    <property name="initialSize" value="${initialSize}" />
11.    <!-- 连接池的最大值 -->
12.    <property name="maxActive" value="${maxActive}" />
13.    <!-- 最大空闲值.当经过一个高峰时间后, 连接池可以慢慢将已经用不到的连接慢慢释放一部分, 一直减少到maxI
14.    <property name="maxIdle" value="${maxIdle}" />
15.    <!-- 最小空闲值.当空闲的连接数少于阀值时, 连接池就会预申请去一些连接, 以免洪峰来时来不及申请 -->
16.    <property name="minIdle" value="${minIdle}" />
17. </bean>
18. <bean id="txManager"
19.     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
20.     <property name="dataSource" ref="dataSource" />
21. </bean>
22. <!-- 采用@Transactional注解方式使用事务 -->
23. <!-- <tx:annotation-driven transaction-manager="txManager" /> -->
24. <!-- 采用xml方式配置事务 -->
25. <aop:config>
26.     <!--
27.         定义切入点
28.         自定义切入点transactionPointcut, 对com.lynzabo.service下所有类和子类的任意方法进行监控
29.     -->
30.     <aop:pointcut id="transactionPointcut" expression="execution(* com.lynzabo.service..*"
31.     <!--
32.         定义一个通知器
33.         引用txAdvice, 引入切入点transactionPointcut -->
34.     <aop:advisor advice-ref="txAdvice" pointcut-ref="transactionPointcut" />
```

```

35.    </aop:config>
36.    <!--
37.        定义通知
38.        使用txManager事务
39.        -->
40.    <tx:advice id="txAdvice" transaction-manager="txManager">
41.        <tx:attributes>
42.            <!-- 以get开头的方法，都不支持事务 -->
43.            <tx:method name="get*" read-only="true" propagation="NOT_SUPPORTED" />
44.            <!-- 其他方法，使用默认的需要事务，使用方法和注解方法一样，属性名都一样 -->
45.            <tx:method name="*" />
46.        </tx:attributes>
47.    </tx:advice>

```

在@Transactional上可以指定属性，和xml里配置aop一样：

propagation 可选的事务传播行为设置

isolation 可选的事务隔离级别设置

readOnly 读写或只读事务，默认为读写

timeout 事务超时时间设置

rollbackFor 导致事务回滚的异常类数组

rollbackForClassName 导致事务回滚的异常类名字数组

noRollbackFor 不会导致事务回滚的异常类数组

noRollbackForClassName 不会导致事务回滚的异常类名字数组

事务传播行为指：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：

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。这是默认值。

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。

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。

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。

PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

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。

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。

隔离级别指：若干个并发的事务之间的隔离程度。定义了五个表示隔离级别的常量：

ISOLATION_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。

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，不可重复读和幻读，因此很少使用该隔离级别。

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。

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。该级别可以防止脏读和不可重复读。

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。

SpringMVC

@Controller注解表示该类扮演控制器的角色。

@RequestMapping注解用于将URL映射到任何一个类或者一个特定的处理方法上。

Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request对象内容进行解析成方法形参，将响应数据和页面封装成 ModelAndView 对象，最后又将模型数据通过request对象传输到页面。Jsp视图解析器默认使用jstl。springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过滤器。

Spring框架中使用到了大量的设计模式，下面列举了比较有代表性的：

代理模式—在AOP和remoting中被用的比较多。

单例模式—在spring配置文件中定义的bean默认为单例模式。

模板方法—用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

前端控制器—Spring提供了DispatcherServlet来对请求进行分发。

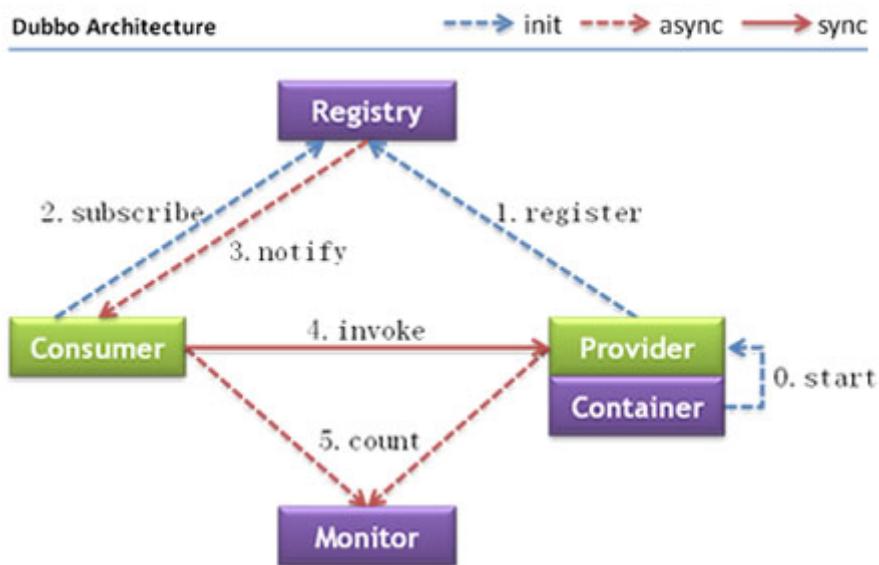
依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念。

工厂模式—BeanFactory用来创建对象的实例。

Spring Boot/Spring Cloud

Dubbo、Dubbox

Dubbo架构



节点角色说明：

- **Provider**: 暴露服务的服务提供方。
- **Consumer**: 调用远程服务的服务消费方。
- **Registry**: 服务注册与发现的注册中心。
- **Monitor**: 统计服务的调用次数和调用时间的监控中心。
- **Container**: 服务运行容器。

调用关系说明：

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示

服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销

服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销

注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外

注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者

注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表

注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

注册中心可以为：

Zookeeper注册中心、Redis注册中心、Multicast注册中心（依赖于网络拓普和路由，跨机房有风险）。

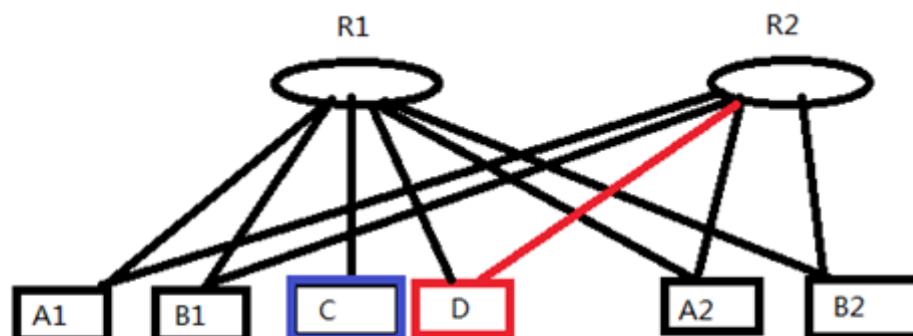
高级用法：

- 消费方直连提供者

```
1. <dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dubbo://localhost:28890" />
```

调试时候使用，这样不影响其他人

- 提供者只订阅
- A服务开发中，A依赖reference于B,C,D服务，我配置A服务只订阅，它能调别人服务，但不注册到注册中心，别人看不到这个服务，消费者直接可以使用上面的直连连接该服务，方便调试。
- 只注册



- 1、A服务和B服务需要依赖于D服务（需要D服务在R1和R2上都注册）
- 2、D服务需要依赖于C服务（D服务只需要订阅R1中的C服务）

A服务和B服务同时依赖于D服务，需要D服务在R1和R2上都注册，让别人调用。D服务依赖于C服务，C服务只在ZK1上注册，这时我们让D服务同时注册和订阅（要调用C）ZK1，对ZK2只注册服务。这个场景经常在容灾，多ZK上会用到。

```
1. <dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />
2. <!--只注册-->
3. <dubbo:registry id="qdRegistry" address="10.20.141.150:9090" subscribe="false" />
```

- Dubbo在安全机制方面是如何解决的

Dubbo通过Token令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo还提供服务黑白名单，来控制服务所允许的调用方。

Dubbox

Dubbo提供了多种均衡策略，缺省为random随机调、还有轮询、一致性Hash、最少活跃调用数。

dubbox是当当网对原阿里dubbo2.x的升级，并且兼容原有的dubbox。其中升级了zookeeper和spring版本，并且支持restfull风格的远程调用。

dubbox的新特性介绍：

- 支持REST风格远程调用 (HTTP + JSON/XML): 基于非常成熟的JBoss RestEasy框架。另外，REST调用也达到了比较高的性能，在基准测试下，HTTP + JSON与Dubbo 2.x默认的RPC协议（即TCP + Hessian2二进制序列化）之间只有1.5倍左右的差距。
- 支持基于Kryo和FST的Java高效序列化实现：基于当今比较知名的Kryo和FST高性能序列化库，为Dubbo默认的RPC协议添加新的序列化实现，并优化调整了其序列化体系，比较显著的提高了Dubbo RPC的性能，详见文档中的基准测试报告。
- 支持基于Jackson的JSON序列化：为Dubbo默认的RPC协议添加新的JSON序列化实现。
- 支持基于嵌入式Tomcat的HTTP remoting体系：用以逐步取代Dubbo中旧版本的嵌入式Jetty，可以显著的提高REST等的远程调用性能，并将Servlet API的支持从2.5升级到3.1。
- 升级Spring：将dubbo中Spring由2.x升级到目前最常用的3.x版本，减少版本冲突带来的麻烦。
- 升级ZooKeeper客户端：将dubbo中的zookeeper客户端升级到最新的版本，以修正老版本中包含的bug。
- 支持完全基于Java代码的Dubbo配置：基于Spring的Java Config，实现完全无XML的纯Java代码方式来配置dubbo

注：dubbox和dubbo 2.x是兼容的，没有改变dubbo的任何已有的功能和配置方式（除了升级了spring之类的版本）

我的测试工程代码：

matrix-service-template

spring-context.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xmlns:context="http://www.springframework.org/schema/context"
5.         xmlns:aop="http://www.springframework.org/schema/aop"
6.         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springf
7.
8.         <description>服务Spring全局配置</description>
9.
10.        <!-- 采用注释的方式配置bean -->
11.        <context:annotation-config/>
12.        <!-- 配置要扫描的包 -->
```

```

13. <context:component-scan base-package="com.le.matrix.template"/>
14. <!-- 读入配置属性文件 -->
15. <context:property-placeholder location="classpath:db.properties,classpath:config.prop
16. <!--<context:property-placeholder location="classpath:public_system.properties,classpath
17. <!-- 读入配置属性文件 -->
18. <!--<bean id="propertyConfigurer"
19.     class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
20.     <!--<property name="location" value="classpath:jdbc.properties" /-->
21. </bean>-->
22. <!-- proxy-target-class默认"false"，更改为"ture"使用CGLib动态代理 -->
23. <aop:aspectj-autoproxy proxy-target-class="true"/>
24.
25. <import resource="classpath:spring/spring-mybatis.xml"/>
26. <import resource="classpath:spring/spring-tx.xml"/>
27. <import resource="classpath*:spring/dubbo/spring-*.xml"/>
28. <!--<import resource="classpath*:spring/dubbo/spring-*.xml"/-->
29. </beans>

```

spring-dubbo-provider.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
5.         xsi:schemaLocation="http://www.springframework.org/schema/beans
6.                             http://www.springframework.org/schema/beans/spring-beans.xsd
7.                             http://code.alibabatech.com/schema/dubbo
8.                             http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
9. <!-- 提供方应用信息，用于计算依赖关系 -->
10. <dubbo:application name="matrix-service-template" owner="matrix" organization="le"/>
11. <!-- 使用zookeeper注册中心暴露服务地址 -->
12. <dubbo:registry address="${dubbo.registry.zkAddr}"/>
13. <!--自动扫描该包路径，省去每个都配合dubbo:service-->
14. <dubbo:annotation package="com.le.matrix.template.service"/>
15. <!-- dubbo协议定义 -->
16. <dubbo:protocol name="dubbo" port="${dubbo.protocol.dubbo}"/>
17.
18. <!--使用内置tomcat服务器 可以省去webapp里面内容 !!!与下面二选一-->
19. <!--如果直接Main启动/junit启动（一般测试时候用），使用下面配置 -->
20. <!--缺省配置-->
21. <!--<dubbo:protocol name="rest" port="${dubbo.protocol.rest}" threads="500" contextpa
22.             /-->
23. <!--使用内置tomcat服务器 可以省去webapp里面内容-->
24. <!--自定义配置-->
25. <!--<dubbo:protocol name="rest" port="${dubbo.protocol.rest}" threads="500" contextpa
26.             extension="com.le.matrix.template.extension.tomcat.TraceInterceptor,
27.                         com.le.matrix.template.extension.tomcat.TraceFilter,
28.                         com.le.matrix.template.extension.tomcat.ClientTraceFilter,
29.                         com.le.matrix.template.extension.tomcat.DynamicTraceBinding,
30.                         com.le.matrix.template.extension.tomcat.CustomExceptionMapper,
31.                         com.alibaba.dubbo.rpc.protocol.rest.support.LoggingFilter"/-->
32. <!--如果使用外部jetty服务器 !!!与上面二选一-->
33. <!-- rest协议定义：
34.         port: 和服务器端口保持一致
35.         contextpath: 和web服务器web.xml定义保持一致
36.         server: 默认servlet，在没有在web服务器中运行的时候，可以写tomcat等由应用启动一个web服务器-->
37. <dubbo:protocol name="rest" port="${dubbo.protocol.rest}" contextpath="services" serv
38.
39. <!--暴露服务列表-->
40. <!--关于service属性配置看http://dubbo.io/User+Guide-zh.htm#UserGuide-zh-%253Cdubbo%253A
41. <!--HelloWorldFacade服务

```

```

42.      interface    服务接口名
43.      ref         服务对象实现引用
44.      retries     远程服务调用重试次数, 默认为2, 不包括第一次调用, 不需要重试请设为0
45.      protocol   协议, 默认为上面定义的所有协议
46.      -->
47.      <dubbo:service interface="com.le.matrix.template.facade.HelloWorldFacade" ref="hello"
48.                      retries="0" protocol="dubbo"/>
49.      <!--HelloWorld2Facade服务
50.          interface    接口名
51.          ref         服务对象实现引用
52.          retries     远程服务调用重试次数, 默认为2, 不包括第一次调用, 不需要重试请设为0
53.          protocol   协议, 默认为上面定义的所有协议
54.          timeout     远程服务调用超时时间
55.          connections 对每个提供者的最大连接数, rmi、http、hessian等短连接协议表示限制连
56.          validation  是否启用JSR303标准注解验证, 如果启用, 将对方法参数上的注解进行校验
57.          group       服务分组, 当一个接口有多个实现, 可以用分组区分
58.          -->
59.      <dubbo:service interface="com.le.matrix.template.facade.HelloWorld2Facade" ref="hello"
60.                      timeout="2000" connections="100" validation="true"/>
61.      <!--HelloWorld3Facade服务
62.          interface    接口名
63.          ref         服务对象实现引用
64.          retries     远程服务调用重试次数, 默认为2, 不包括第一次调用, 不需要重试请设为0
65.          protocol   协议, 默认为上面定义的所有协议
66.          timeout     远程服务调用超时时间
67.          connections 对每个提供者的最大连接数, rmi、http、hessian等短连接协议表示限制连
68.          validation  是否启用JSR303标准注解验证, 如果启用, 将对方法参数上的注解进行校验
69.          -->
70.      <dubbo:service interface="com.le.matrix.template.facade.HelloWorld3Facade" ref="hello"
71.                      timeout="2000" connections="100" validation="true"/>
72.      <!--HelloWorld5Facade服务
73.          interface    接口名
74.          ref         服务对象实现引用
75.          retries     远程服务调用重试次数, 默认为2, 不包括第一次调用, 不需要重试请设为0
76.          protocol   协议, 默认为上面定义的所有协议
77.          timeout     远程服务调用超时时间
78.          connections 对每个提供者的最大连接数, rmi、http、hessian等短连接协议表示限制连
79.          validation  是否启用JSR303标准注解验证, 如果启用, 将对方法参数上的注解进行校验
80.          version     服务版本, 建议使用两位数字版本, 如: 1.0, 通常在接口不兼容时版本号才需
81.          -->
82.      <dubbo:service interface="com.le.matrix.template.facade.HelloWorld5Facade" ref="hello"
83.                      timeout="2000" connections="100" validation="true" version="10.0"/>
84.
85.      <dubbo:service interface="com.le.matrix.template.facade.UserFacade" ref="userService">
86.
87.          <!--<dubbo:reference interface="com.xxx.XxxService">
88.              <dubbo:method name="findXxx" timeout="3000" retries="2" />
89.          </dubbo:reference>-->
90.
91.          <!--TODO 需要再次验证
92.              1.验证dubbo/rest传对象
93.              2.验证传对象validate
94.              3.验证服务端抛出异常, 客户端dubbo/rest接收异常
95.              4.验证服务端异常统一处理, 并返回对应http状态码
96.
97.          -->
</beans>
```

HelloWorld2ServiceImpl

```

1. package com.le.matrix.template.service.impl;
2.
```

```

3. import com.alibaba.dubbo.rpc.protocol.rest.support.ContentType;
4. import com.le.matrix.template.facade.HelloWorld2Facade;
5. import org.slf4j.Logger;
6. import org.slf4j.LoggerFactory;
7. import org.springframework.stereotype.Service;
8.
9. import javax.ws.rs.*;
10. import javax.ws.rs.core.MediaType;
11.
12. /**
13.  * Created by linzhanbo on 2016/10/10.
14. */
15. @Service("helloWorld2Service")
16. //Rest定义、异常处理详看 RestEasy http://dangdangdotcom.github.io/dubbox/rest.html
17. //如统一处理使用javax.ws.rs, 若异常 http://redhacker.iteye.com/blog/1924071
18. //REST路径
19. @Path("helloWorld2")
20. //标注可接受请求的MIME媒体类型
21. @Consumes({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
22. //标注返回的MIME媒体类型
23. @Produces({ContentType.APPLICATION_JSON_UTF_8, ContentType.TEXT_XML_UTF_8})
24. public class HelloWorld2ServiceImpl implements HelloWorld2Facade {
25.     private Logger logger = LoggerFactory.getLogger(HelloWorld2ServiceImpl.class);
26.
27.     @GET
28.     @Path("{id}")
29.     @Override
30.     public String sayHi(@PathParam("id") String id) {
31.         String hiStr = "Hello," + id;
32.         logger.debug("sayhello to {}", id);
33.         return hiStr;
34.     }
35. }

```

HelloWorldServiceImpl

```

1. package com.le.matrix.template.service.impl;
2.
3. import com.le.matrix.template.facade.HelloWorldFacade;
4. import org.slf4j.Logger;
5. import org.slf4j.LoggerFactory;
6. import org.springframework.stereotype.Service;
7.
8. /**
9.  * Created by linzhanbo on 2016/10/10.
10. */
11. @Service("helloWorldService")
12. public class HelloWorldServiceImpl implements HelloWorldFacade {
13.     private Logger logger = LoggerFactory.getLogger(HelloWorldServiceImpl.class);
14.
15.     @Override
16.     public String sayHello(String username) {
17.         String helloStr = "Hello," + username;
18.         logger.debug("sayhello to {}", username);
19.         return helloStr;
20.     }
21. }

```

spring-dubbo-consumer.xml

```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://www.springframework.org/schema/beans
5.      http://www.springframework.org/schema/beans/spring-beans.xsd">
6.
7.    <!-- 调用HelloWorldFacade服务 -->
8.    <!!--<dubbo:reference interface="com.le.matrix.template.facade.HelloWorldFacade" id="helloWorldService" check="false" />-->
9.
10.   </beans>

```

matrix-web-template

spring-dubbo-consumer.xml

```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
4.    xmlns="http://www.springframework.org/schema/beans"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/spring-beans.xsd
7.      http://code.alibabatech.com/schema/dubbo
8.      http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
9.    <!-- 提供方应用信息, 用于计算依赖关系 -->
10.   <dubbo:application name="matrix-web-redis" owner="matrix" organization="le"/>
11.   <!-- 使用zookeeper注册中心暴露服务地址 -->
12.   <dubbo:registry address="${dubbo.registry.zkAddr}" />
13.   <!--服务引用列表-->
14.   <!--关于reference属性配置看http://dubbo.io/User+Guide-zh.htm#UserGuide-zh-%253Cdubbo%253E-->
15.   <!-- 调用HelloWorldFacade服务
16.     interface 服务接口名
17.     id         服务引用BeanId
18.     check      启动时检查提供者是否存在, true报错, false忽略
19.   -->
20.   <dubbo:reference interface="com.le.matrix.template.facade.HelloWorldFacade" id="helloWorldService" check="false"/>
21.   <!-- 调用HelloWorldFacade服务
22.     interface 服务接口名
23.     id         服务引用BeanId
24.     check      启动时检查提供者是否存在, true报错, false忽略
25.     url        点对点直连服务提供者地址, 将绕过注册中心          调试时候经常使用, 线上不要用
26.   -->
27.   <dubbo:reference interface="com.le.matrix.template.facade.HelloWorld3Facade" id="helloWorld3Service" check="false" url="dubbo://localhost:8888/services/" />
28.   <!-- 调用HelloWorld5Facade服务
29.     interface 服务接口名
30.     id         服务引用BeanId
31.     check      启动时检查提供者是否存在, true报错, false忽略
32.     group     服务分组, 当一个接口有多个实现, 可以用分组区分, 必需和服务提供方一致
33.     retries   远程服务调用重试次数, 不包括第一次调用, 不需要重试请设为0
34.     version   服务版本, 与服务提供者的版本一致
35.   -->
36.   <!--注意: 连接rest的服务时候, rs.ws定义必须在接口中定义, 否则报错You must use at least one, but none was found-->
37.   <!--如何定义详细看com.le.matrix.template.facade.HelloWorld5Facade-->
38.   <dubbo:reference interface="com.le.matrix.template.facade.HelloWorld5Facade" id="helloWorld5Service" check="false" version="10.0" />
39.   <!-- 调用UserFacade服务
40.     interface 服务接口名
41.     id         服务引用BeanId
42.     check      启动时检查提供者是否存在, true报错, false忽略
43.   -->

```

```
44.     protocol      指定使用协议 如果服务提供多个协议支持，消费端必须指定使用协议
45.     -->
46.     <dubbo:reference interface="com.le.matrix.template.facade.UserFacade" id="userService"
47.                           protocol="dubbo"/>
48.   </beans>
```

直接使用对Rest封装HTTP client测试 RestClient

```
1. package com.le.matrix.template;
2.
3. import javax.ws.rs.client.Client;
4. import javax.ws.rs.client.ClientBuilder;
5. import javax.ws.rs.client.Entity;
6. import javax.ws.rs.client.WebTarget;
7. import javax.ws.rs.core.MediaType;
8. import javax.ws.rs.core.Response;
9.
10. /**
11.  * Created by linzhanbo on 2016/10/10.
12. */
13. public class RestClient {
14.     public static void main(String[] args) {
15.         getUser("http://localhost:8888/services/helloWorld2/aaa.json");
16.     }
17.
18.     private static void getUser(String url) {
19.         System.out.println("Getting user via " + url);
20.         Client client = ClientBuilder.newClient();
21.         WebTarget target = client.target(url);
22.         Response response = target.request().get();
23.         try {
24.             if (response.getStatus() != 200) {
25.                 throw new RuntimeException("Failed with HTTP error code : " + response.get
26.             }
27.             System.out.println("Successfully got result: " + response.readEntity(String.c
28.         } finally {
29.             response.close();
30.             client.close();
31.         }
32.     }
33.
34.     private static void registerUser(String url, MediaType mediaType) {
35.         Client client = ClientBuilder.newClient();
36.         WebTarget target = client.target(url);
37.         String result = new String();
38.         Response response = target.request().post(Entity.entity(result, mediaType));
39.
40.         try {
41.             if (response.getStatus() != 200) {
42.                 throw new RuntimeException("Failed with HTTP error code : " + response.get
43.             }
44.             System.out.println("Successfully got result: " + response.readEntity(String.c
45.         } finally {
46.             response.close();
47.             client.close();
48.         }
49.     }
50.     /*private static void registerUser(String url, MediaType mediaType) {
51.         System.out.println("Registering user via " + url);
52.         User user = new User(1L, "larrypage");
53.         Client client = ClientBuilder.newClient();
```

```
54.     WebTarget target = client.target(url);
55.     Response response = target.request().post(Entity.entity(user, mediaType));
56.
57.     try {
58.         if (response.getStatus() != 200) {
59.             throw new RuntimeException("Failed with HTTP error code : " + response.get
60.         }
61.         System.out.println("Successfully got result: " + response.readEntity(String.c
62.     } finally {
63.         response.close();
64.         client.close();
65.     }
66. }
67.
68. private static void getUser(String url) {
69.     System.out.println("Getting user via " + url);
70.     Client client = ClientBuilder.newClient();
71.     WebTarget target = client.target(url);
72.     Response response = target.request().get();
73.     try {
74.         if (response.getStatus() != 200) {
75.             throw new RuntimeException("Failed with HTTP error code : " + response.get
76.         }
77.         System.out.println("Successfully got result: " + response.readEntity(String.c
78.     } finally {
79.         response.close();
80.         client.close();
81.     }
82.     }*/
83. }
```

二 In Go

Golang基础

Go常用命令简介

go get: 获取远程包 (需提前安装 git或hg)

go run: 直接运行程序

go build: 测试编译, 检查是否有编译错误

go fmt: 格式化源码 (部分IDE在保存时自动调用)

go install: 编译包文件并编译整个程序

go test: 运行测试文件

go doc: 查看文档 (CHM手册)

```
1. // 当前程序的包名 使用包名唯一确定程序
2. package main
3.
4. //导入其他的包      使用双引号括起来包名
5. //import "fmt"
6. //import "os"
7. //import "time"
8. //import "strings"
9. //如果不想输入这么多可以在()里面写入全部
10. import (
11.     "os"
12.     "strings"
13.     "time"
14.     std "fmt"    //别名调用
15.     . "strconv" //省略调用
16.     _ "hello/imp" //导入该包时, 包下init()函数会被执行
17. )
18.
19. //常量的定义 大写的常量、变量、类型、接口、结构或函数就是public, 可以被外部使用的。
20. const PI = 3.14
21. //多个常量定义
22. const (
23.     PI2    = 3.14
24.     WIDTH  = 10
25. )
```

```
27.  
28. /*  
29. Go语言中，使用 大小写 来决定该 常量、变量、类型、接口、结构或函数 是否可以被外部包所调用：  
30. 根据约定，  
31. 函数名首字母 小写，或使用_ 即为private  
32. 函数名首字母 大写 即为public  
33. */  
34.  
35.  
36. //全局变量的声明与赋值  
37. var name = "gopher"  
38. //一般类型声明 type 一般类型名称 基本类型 这里newType就是int类型  
39. type newType int  
40.  
41. //多个一般类型定义  
42. type (  
43.     typeor2 int  
44.     vvv2    float32  
45. )  
46.  
47.  
48. //结构的声明  
49. type gopher struct {  
50.     //结构体  
51. }  
52.  
53. //多个结构声明  
54. type (  
55.     sss2 struct{}  
56.     vvv struct{}  
57. )  
58.  
59. //接口的声明  
60. type golang interface{  
61.  
62. //多个接口声明  
63. type (  
64.     aaaa interface{}  
65.     bbbb interface{}  
66.     cccc struct{}  
67. )  
68.  
69.  
70. //枚举使用  
71. const (  
72.     a = 'A'  
73.     b  
74.     c = iota //2     iota是常量的计数器，从0开始，组中每定义1个常量自动  
75.     //递增1，每遇到一个const关键字，iota就会重置为0  
76.     //如果希望后面的值也自增，对前一个使用iota  
77.     d //3  
78.     e //4  
79.     f = 'B'  
80.     g = iota //6  
81. )  
82. const (  
83.     h = iota //0  
84. )  
85.  
86.  
87. //由 main 函数作为程序入口点启动
```

```
88. func main() {
89.     std.Println("package new name")
90.
91.     var i int8 //变量声明
92.     i = 10 //变量的赋值
93.     //变量声明的同时又赋值
94.     var j int64 = 100
95.     var c = 100 //上行的格式可省略变量类型，由系统推断
96.     d := 456 //变量声明与赋值的最简写法，要是想省去var
97.     var a,b,c,d = 1,2,3,4
98.
99.     var a float32 = 100.1
100.    fmt.Println(a) //100.1
101.    b := int(a) //Go中不存在隐式转换，所有类型转换必须显式声明
102.    fmt.Println(b) //100
103.
104.    var a int = 65
105.    b := string(a)
106.    fmt.Println(b) //A
107.    c := /*strconv.*/Itoa(a)
108.    fmt.Println(c) //"65"
109.    i,err := /*strconv.*/ParseInt("-42",10,64)
110.    fmt.Println(i) // -42
111.
112.
113.    a := 1
114.    var p *int = &a
115.    fmt.Println(p) //0xf840038020 输出p指针地址
116.    fmt.Println(*p) //1 输出p指针指向的值
117. /*
118. Go虽然保留了指针，但与其它编程语言不同的是，在Go当中不支持指针运算以及"->"运算符，而直接采用"."选择符来操
119. 作符"&"取变量地址，使用"*"通过指针间接访问目标对象
120. 默认值为 nil 而非 NULL
121. */
122.
123.    a := 1
124.    //在Go当中，++ 与 -- 是作为语句而并不是作为表达式
125.    //只能放在变量 右边，不能放在变量 左边
126.    a++
127.    a--
128.    //fmt.Println(a++) //不可以这样
129.
130.    //支持一个初始化表达式，;隔开，a只在if内作用
131.    /*if a := 1; a > 1 {
132.    }*/
133.
134.
135.    //go利用一个关键字for达到其他语言的for, while,do ...while三种语句作用
136.    /*
137.        a := 1
138.        for {
139.            a++
140.            if a > 3 {
141.                break
142.            }
143.            fmt.Println(a)
144.        }
145.        /*
146.        a := 1
147.        for a <= 3 {
```

```
149.         a++
150.         fmt.Println(a)
151.     }
152.     */
153.     /*
154.     a := 1
155.     for i := 0; i < 3; i++ {
156.         a++
157.         fmt.Println(a)
158.     }
159.     */
160.
161.
162.     //switch
163.     /*
164.     a := 1
165.     switch a { //switch支持表达式
166.     case 0:
167.         fmt.Println("a=0") //不需要写break
168.     case 1:
169.         fmt.Println("a=1")
170.     default:
171.         fmt.Println("None")
172.     }
173.     /*
174.     a := 1
175.     switch { //switch没放判断条件, 在下面各个放
176.     case a >= 0:
177.         fmt.Println("a=0") //不需要写break
178.         fallthrough //让继续执行下面case
179.     case a >= 1:
180.         fmt.Println("a=1")
181.     default:
182.         fmt.Println("None")
183.     }
184.     /*
185.     switch a := 1; { //switch没放判断条件, 在下面各个放
186.     case a >= 0:
187.         fmt.Println("a=0") //不需要写break
188.         fallthrough //让继续执行下面case
189.     case a >= 1:
190.         fmt.Println("a=1")
191.     default:
192.         fmt.Println("None")
193.     }
194.
195.
196.
197.
198. }
199.
200.
201. /*
202. 对于break 标签/continue 标签与goto 标签:
203. goto标签是调整执行位置, 与其他两个语句配合标签的结果不相同, 使用goto标签, 放到循环后面让直接跳出循环。
204. */
205. func main() {
206. LABEL1:
207.     for {
208.         for i := 0; i < 10; i++ {
209.             if i > 3 {
```

```
210.             break LABEL1 //成功跳出n处循环
211.
212.         }
213.     }
214.     fmt.Println("OK")
215. }
216. func main() {
217.     for {
218.         for i := 0; i < 10; i++ {
219.             if i > 3 {
220.                 goto LABEL1 //成功跳出n处循环
221.             }
222.         }
223.     }
224. LABEL1:
225.     fmt.Println("OK")
226. }
227. */
228.
229. func main() {
230.
231.     //数组
232.     //定义数组
233.     //var a [2]int
234.     //定义数组并初始化
235.     //a := [2]int{1, 2} //[1 2]
236.     //a := [2]int{1} //*[1 0]    剩下值不够, 自动补默认值
237.     //如果希望将第20个赋为1
238.     //a := [20]int{19: 1} //*[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
239.     //如果已经知道数组长度, 可以不需要指定长度
240.     //a := [...]int{1, 2, 3, 4, 5} //*[1 2 3 4 5]
241.     //a := [...]int{10: 1, 12: 3} //*[0 0 0 0 0 0 0 1 0 3] 长度为13
242.
243.     //切片
244.     //其本身并不是数组, 它指向底层的数组。作为变长数组的替代方案, 可以关联底层数组的局部或全部
245.     //为引用类型
246.     //可以直接创建或从底层数组获取生成, 使用len()获取元素个数, cap()获取容量
247.     //直接创建: make([]T, len, cap) 其中cap可以省略, 则和len的值相同      len表示存放的元素个数,
248.     //slice声明方法1 和数组区别是没有指定大小
249.     var s1 []int
250.     fmt.Println(s1) //[]
251.     array := [10]int{5: 1}
252.     fmt.Println(array) //*[0 0 0 0 1 0 0 0 0]
253.     //slice 索引为5到最后
254.     s2 := array[5:10] //array[5 6 7 8 9]
255.     fmt.Println(s2) //*[1 0 0 0 0]
256.     //取从指定索引到最后还有两种办法
257.     //slice 方法1
258.     s3 := array[5:len(array)] //array[5 6 7 8 9]
259.     fmt.Println(s3) //*[1 0 0 0 0]
260.     //slice 方法2
261.     s4 := array[5:] //array[5 6 7 8 9]
262.     fmt.Println(s4) //*[1 0 0 0 0]
263.     //slice 我想取数组前5个元素
264.     s5 := array[:5] //array[0 1 2 3 4]
265.     fmt.Println(s5) //*[0 0 0 0 0]
266.
267.     //slice声明方法2
268.     s6 := make([]int, 3, 10) //指定slice 类型为[]int, 包含3个元素,slice初始容量为10, 数组, 这10个
269.     //slice是一个变长的数组,我指定初始容量为10, 即便放了3个元素, 后面再添加元素的时候, 也不需要new新地址
270.     //个连续地址,如果要添加第11个元素, 这时go语言会把这个数组大小变成10*2,20个大小, 到第21个, 变成20*2
```

```
271. //如果你事先知道slice大小，应该make时给足初始容量大小。如果不设置初始容量，则初始为元素个数
272. fmt.Println(len(s6), cap(s6)) //3 10 len打印slice实际长度 cap打印slice容量
273. s7 := make([]int, 3)
274. fmt.Println(len(s7), cap(s7)) //3 3

275.
276. s1 := []int{1, 2, 3, 4, 5, 6, 7}
277. s2 := []int{8, 9, 10}
278. copy(s2, s1)      //将s1复制到s2中
279. fmt.Println(s2) //[[1 2 3]] 只复制了这3位

280.
281. copy(s1[3:5], s2[1:3]) //将s2复制到s1中
282. fmt.Println(s1)        // [1 2 3 9 10 6 7]

283.
284. //slice迭代
285. arr := []string{"ele1", "ele2", "ele3", "ele4", "ele5"}
286. slice := arr[0:] //完全引用数组
287. for index, value := range slice {
288.     fmt.Printf("index=%d,value=%s\n", index, value)
289. }

290.
291.
292. //map
293. //类似其它语言中的哈希表或者字典，以key-value形式存储数据
294. //Key必须是支持==或!=比较运算的类型，不可以是函数、map或slice
295. //Map使用make()创建，支持 := 这种简写方式
296. //make([keyType]valueType, cap)，cap表示容量，可省略，超出容量时会自动扩容，但尽量提供一个合理的
297. //键值对不存在时自动添加，使用delete()删除某键值对
298. //使用 for range 对map和slice进行迭代操作
299. //创建集合
300. var m map[int]string
301. m = map[int]string{}
302. //创建集合2
303. var m2 map[int]string = make(map[int]string)
304. //也可以使用简写办法
305. m3 := make(map[int]string) //也可以指定map初始大小
306. fmt.Println(m)           //map []
307. fmt.Println(m2)          //map []
308. fmt.Println(m3)          //map []

309.
310. m := make(map[int]string)
311. m[1] = "OK" //添加一个键值对
312. m[2] = "OK" //添加一个键值对
313. a := m[1]
314. b := m[3]      //没有取出来就为空""
315. delete(m, 2)   //使用键删除该KV
316. fmt.Println(m) //map [1:OK]
317. fmt.Println(a) //OK
318. fmt.Println(b) //

319.
320. //map是无序的
321. m := map[int]string{
322.     1: "a",
323.     2: "b",
324.     3: "c",
325.     4: "d",
326.     5: "e"}
327. s := make([]int, len(m))
328. i := 0
329. for key, _ := range m {
330.     s[i] = key
331.     i++
```

```
332.
333.     }
334.     fmt.Println(s) //输出多次，发现每次输出的Key的顺序都不同[4 1 2 5 3] [2 4 5 3 1]
335.     //讲sort里一个排序方法
336.     sort.Ints(s)
337.     fmt.Println(s) //[1 2 3 4 5]
338.
339.
340. /*
341. 定义函数规则：
342. 内部函数首字母小写，外部函数首字母大写
343. 如果函数没参数，直接A()，有参数，先参数名，再参数类型
344. 如果函数没有返回值，则直接func A(参数列表) {,
345. 如果有返回值，在括号内写所有要返回的参数类型，如果只有一个返回值，括号可以省去
346. */
347. func A(param1 int, param2 string) (int, string, bool) {
348. }
349.
350. //接收类型相同，不定长长度
351. //f(1,2) f(1,3,4,5)
352. func f(a ...int) {
353.     for i := range a {
354.         fmt.Println(a[i])
355.     }
356.     a[1] = 3
357.     fmt.Println(a) //[1 3]
358. }
359.
360.
361. //匿名函数
362. func main() {
363.     a := func() { //a为这种类型的变量
364.         fmt.Println("func exec")
365.     }
366.     a() //func exec
367. }
368.
369. //go语言也支持闭包
370. func main() {
371.     c := closure(10)
372.     fmt.Println(c(1)) //11
373.     fmt.Println(c(2)) //12
374. }
375. //该函数返回函数类型
376. func closure(x int) func(int) int {
377.     fmt.Printf("%p\n", &x) //0xf840038020 三次地址都一样
378.     return func(y int) int {
379.         fmt.Printf("%p\n", &x) //0xf840038020
380.         return x + y
381.     }
382. }
383.
384. //defer
385. func main() {
386.     fmt.Println("a")
387.     defer fmt.Println("b")
388.     defer fmt.Println("c")
389.     /*
390.         输出顺序为
391.         a
392.         c
```

```
393.         b
394.         defer fmt.Println("b")
395.     */
396. }
397.
398. func main() {
399.     fmt.Println("a")
400.     defer fmt.Println("b")
401.     defer fmt.Println("c")
402.     /*
403.         输出顺序为
404.         a
405.         c
406.         b
407.         defer定义按照调用顺序的相反顺序逐个执行
408.     */
409. }
410.
411.
412.
413. //go语言没有try catch, 取代这种模式使用的panic、recover模式, Panic 可以在任何地方引发, 但recover只有
414.
415. func main() {
416.     A()
417.     B() //B有panic, C函数就不执行了
418.     C()//可以看到A,B执行了, 然后就抛出异常了, C没执行
419. }
420. func A() {
421.     fmt.Println("Func A")
422. }
423. func B() {
424.     panic("Func B")
425. }
426. func C() {
427.     fmt.Println("Func C")
428. }

429.
430.
431. //将程序从panic状态recover到正常状态
432. func main() {
433.     A()
434.     B() //B有panic, C函数就不执行了
435.     C()
436.     /*
437.     Func A
438.     Recover in B
439.     Func C
440.     */
441.     /*
442.     */
443. }
444. func A() {
445.     fmt.Println("Func A")
446. }
447. func B() {
448.     defer func() {
449.         if err := recover(); err != nil {
450.             fmt.Println("Recover in B")
451.         }
452.     }()
453.     panic("Func B")
```

```
454.
455. }
456. func C() {
457.     fmt.Println("Func C")
458.
459.
460. //defer必须在panic之前出现, panic在哪个方法内, 则停掉执行该方法后的所有。但不影响其他方法调用。这样才能捕
461. 
462. func main(){
463.     A()
464.     B()
465.     panic("111")
466.     C()
467.     /*
468.      执行结果:
469.      panic: Func B
470.      Func A
471.
472.      goroutine 1 [running]:
473.      */
474. }
475. func A() {
476.     fmt.Println("Func A")
477. }
478. func B() {
479.     defer func() {
480.         if err := recover(); err != nil {
481.             fmt.Println("Recover in B")
482.         }
483.     }()
484. }
485. func C() {
486.     fmt.Println("Func C")
487. }
488.
489. //修改位置
490.
491. func main(){
492.     A()
493.     B()
494.
495.     C()
496.     /*
497.      执行结果:
498.      Func A
499.      Recover in B
500.      Func C
501.     */
502. }
503. func A() {
504.     fmt.Println("Func A")
505. }
506. func B() {
507.     defer func() {
508.         if err := recover(); err != nil {
509.             fmt.Println("Recover in B")
510.         }
511.     }()
512.     panic("111")
513. }
514. func C() {
```

```
515.         fmt.Println("Func C")
516.     }
517.
518. //或者
519. func main(){
520.     A()
521.
522.     defer func() {
523.         if err := recover(); err != nil {
524.             fmt.Println("Recover in B")
525.         }
526.     }()
527.     B()
528.     C()
529.     /*
530.      执行结果:
531.      Func A
532.      Recover in B
533.      */
534. }
535. func A() {
536.     fmt.Println("Func A")
537. }
538. func B() {
539.     panic("111")
540. }
541. func C() {
542.     fmt.Println("Func C")
543. }
544.
545. //练习，看输出什么
546. package main
547.
548. import "fmt"
549.
550. func main(){
551.     defer func(){ // 必须要先声明defer, 否则不能捕获到panic异常
552.         fmt.Println("c")
553.         if err:=recover();err!=nil{
554.             fmt.Println(err) // 这里的err其实就是panic传入的内容, 55
555.         }
556.         fmt.Println("d")
557.     }()
558.     f()
559. }
560.
561. func f(){
562.     fmt.Println("a")
563.     panic(55)
564.     fmt.Println("b")
565.     fmt.Println("f")
566. }
567. /*
568.  输出结果:
569.  a
570.  c
571.  55
572.  d
573.  exit code 0, process exited normally.
574. */
575.
```

```
576.  
577.  
578. //结构struct  
579. type person struct {  
580.     Name string  
581.     Age  int  
582. }  
583. func main() {  
584.     a := person{}  
585.     a.Name = "joe"  
586.     a.Age = 13  
587.     fmt.Println(a) //{joe 13}  
588. }  
589. //go没有构造方法，但是也提供了简单赋值  
590. func main() {  
591.     a := person{  
592.         Name: "joe",  
593.     }  
594.     a.Age = 13  
595.     fmt.Println(a) //{joe 13}  
596.  
597.     b := person{  
598.         Name: "joe",  
599.         Age:  13,  
600.     }  
601.     fmt.Println(b) //{joe 13}  
602. }  
603.  
604. //go传递的是引用  
605. type person struct {  
606.     Name string  
607.     Age  int  
608. }  
609. func main() {  
610.     a := person{  
611.         Name: "joe",  
612.         Age:  13,  
613.     }  
614.     fmt.Println(a) //{joe 13}  
615.     A(a)  
616.     fmt.Println(a) //{joe 13}  
617.     B(&a)  
618.     fmt.Println(a) //{joe 21}  
619. }  
620. func A(per person) {  
621.     per.Age = 20  
622.     fmt.Println("perA", per) //perA {joe 20}  
623. }  
624. func B(per *person) {  
625.     per.Age = 21  
626.     fmt.Println("perB", *per) //perB {joe 21}  
627. }  
628.  
629. //习惯定义对象直接定义成指针类型，go对指针的使用直接都是.，这样定义方便以后修改  
630. func main() {  
631.     a := &person{  
632.         Name: "joe",  
633.         Age:  13,  
634.     }  
635.     fmt.Println(a) //&{joe 13}  
636.     B(a)
```

```
637.     fmt.Println(a) //&{joe 21}
638.
639. }
640. func B(per *person) {
641.     per.Age = 21
642.     fmt.Println("perB", per) //perB &{joe 21}
643. }
644.
645. //匿名结构
646. func main() {
647.     per := &struct { //定义结构
648.         Name string
649.         Age int
650.     }{ //初始化结构
651.         Name: "Joe",
652.         Age: 12,
653.     }
654.     fmt.Println(per)
655. }
656.
657. //匿名字段
658. type Person struct {
659.     //匿名字段
660.     string
661.     int
662. }
663. func main() {
664.     a := Person("joe", 19) //初始化时按照匿名字段定义顺序赋值
665.     fmt.Println(a)
666. }
667. //匿名结构
668. type Person struct {
669.     Name    string
670.     Age     int
671.     Contact struct {
672.         /*
673.             Phone string
674.             City   string
675.         */
676.         //类型相同，可以省略之写一次类型
677.         Phone, City string
678.     }
679. }
680. func main() {
681.     person := Person{
682.         Name: "Joe",
683.         Age: 19,
684.     }
685.     //Contact是个类型
686.     person.Contact.Phone = "2143423"
687.     person.Contact.City = "Beijing"
688.     fmt.Println(person)
689. }
690. type A struct {
691.     B
692.     C
693. }
694. type B struct {
695.     Name string
696.     Sex  int
697. }
```

```
698. type C struct {
699.     Name string
700. }
701. func main() {
702.     a := A{B: B{Name: "B", Sex: 12}, C: C{Name: "C"}}
703.     fmt.Println(a.Sex, a.B.Sex) //12 12
704.     //fmt.Println(a.Name)    //编译器无法确认获取B还是C的，这时会报错
705.     fmt.Println(a.B.Name, a.C.Name) //B C
706. }
707.
708.
709. //go中method
710. //Go 中虽没有class，但依旧有method。通过显示说明receiver来实现与某个类型的组合，如扩展系统提供的哪些类
711. type A struct {
712.     Name string
713. }
714. type B struct {
715.     Name string
716. }
717. func main() {
718.     a := A{}
719.     //调用A结构的方法
720.     a.Print()
721. }
722. //定义接收者规则
723. //还是func关键字 (变量 接收者类型) 方法名称(方法参数) {}
724. //这样这个方法就和A结构链在了一起
725. func (a A) Print() {
726.     fmt.Println("A")
727. }
728.
729. //别名定义
730. type TZ int
731. func main() {
732.     var a TZ
733.     a.Print() //TZ
734. }
735. func (a *TZ) Print() {
736.     fmt.Println("TZ")
737. }
738.
739. //如上面，如果大家对int类型由一个更高级的操作，更高级的封装，就可以把int先作为底层类型，然后在对自定义类型
740.
741.
742. //Method Value方式和Method Expression方式一样
743. type TZ int
744. func main() {
745.     var a TZ
746.     //这是Method Value方式
747.     a.Print() //TZ
748.     //receiver是指针类型
749.     //这是Method Expression方式 这两种方式一样
750.     (*TZ).Print(&a) //TZ
751. }
752. func (a *TZ) Print() {
753.     fmt.Println("TZ")
754. }
755.
756.
757. //接口
```

```
759. //只要某个类型拥有该接口的所有方法签名，即算实现该接口，无需显示
760. //定义接口
761. type USB interface {
762.     Name() string
763.     Connect()
764. }
765. //定义实现接口的结构
766. type PhoneConnector struct {
767.     name string
768. }
769. //定义结构实现接口方法
770. func (pc PhoneConnector) Name() string {
771.     return pc.name
772. }
773. func (pc PhoneConnector) Connect() {
774.     fmt.Println("Connect:", pc.name)
775. }
776. func main() {
777.     //可以看到PhoneConnector实现了USB接口
778.     var a USB
779.     a = PhoneConnector{"PhoneConnector"}
780.     a.Connect() //Connect: PhoneConnector
781.     Disconnect(a) //Disconnected.
782. }
783. func Disconnect(usb USB) {
784.     fmt.Println("Disconnected.")
785. }
786.
787. //定义接口
788. type USB interface {
789.     Name() string
790.     Connector //嵌入接口
791. }
792. type Connector interface {
793.     Connect()
794. }
795.
796.
797.
798. //其他语言中任何类都有父类Object，go没有继承概念，但是也存在顶级父类
799. //go语言不需要你显示指定实现了哪个接口，而只需要实现接口的所有方法，只要实现类实现的方法和接口定义方法签名
800. type empty interface {
801. }
802. //定义接口
803. type USB interface {
804.     Name() string
805.     Connector //嵌入接口
806. }
807. type Connector interface {
808.     Connect()
809. }
810. //定义实现接口的结构
811. type PhoneConnector struct {
812.     name string
813. }
814. //定义结构实现接口方法
815. func (pc PhoneConnector) Name() string {
816.     return pc.name
817. }
818. func (pc PhoneConnector) Connect() {
819.     fmt.Println("Connect:", pc.name)
```

```

820.
821. }
822. func main() {
823.     //可以看到PhoneConnector实现了USB接口
824.     var a USB
825.     a = PhoneConnector{"PhoneConnector"}
826.     a.Connect() //Connect: PhoneConnector
827.     Disconnect(a) //Disconnected:PhoneConnector
828. }
829. func Disconnect(usb USB) {
830.     //判断usb到底是哪个具体类型方法
831.     if pc, ok := usb.(PhoneConnector); ok {
832.         fmt.Println("Disconnected:", pc.name)
833.         return
834.     }
835.     fmt.Println("Unknown device.")
836. }
837. func Disconnect(usb interface{}) { //该办法可以传递任何的实现类进来，但是如果你明确实现那个接口，最好
838.     //判断usb到底是哪个具体类型方法，使用这个办法代码有点复杂，go提供更简便办法
839.     /*
840.         if pc, ok := usb.(PhoneConnector); ok {
841.             fmt.Println("Disconnected:", pc.name)
842.             return
843.         }
844.     */
845.     //go提供的简便的判断类型方法      type, 让系统去判断类型
846.     switch v := usb.(type) {
847.     case PhoneConnector:
848.         fmt.Println("Disconnected:", v.name)
849.     default:
850.         fmt.Println("Unknown device.")
851.     }
852. }

```

go专门提供了reflect库专门用来实现反射的。

并发主要由切换时间片来实现“同时”运行，而并行则是直接利用多核实现多线程的运行，但 Go 可以设置使用核数，以发挥多核计算机的能力。

Goroutine 奉行通过通信来共享内存，而不是共享内存来通信。

```

1. func main() {
2.     //运行goroutine
3.     go Go()
4.     time.Sleep(2 * time.Second) //休息2s，注意请使用go提供的表示单位
5. }
6. func Go() {
7.     fmt.Println("Go Go Go!!!")
8. }
9.
10.
11.
12. func main() {
13.     //让主线程等待子线程执行完毕在退出
14.     c := make(chan bool)
15.     //运行goroutine
16.     go func() {
17.         fmt.Println("Go Go Go!!!")
18.         c <- true //上面channel定义成bool类型，所以这里返回bool true/false都可以

```

```

19. }()
20. <-c //当执行goroutine时，就会阻塞在这一行，等子线程内发出结束消息
21. //程序非常简单，退出程序时候就自动释放了，不需要close关闭
22. }

23.

24.

25. //可以使用 for range 来迭代不断操作 channel
26. func main() {
27.     //让主线程等待子线程执行完毕在退出
28.     c := make(chan bool)
29.     //运行goroutine
30.     go func() {
31.         fmt.Println("Go Go Go!!!")
32.         c <- true //上面channel定义成bool类型，所以这里返回bool true/false都可以
33.         time.Sleep(2 * time.Second)
34.         c <- false
35.         close(c)
36.     }()
37.     for v := range c {
38.         fmt.Println(v)
39.     }
40.     fmt.Println("Exit!")
41.     /*
42.      首先主线程在迭代处等待，子线程输出了GO GO GO!!!，然后往channel中放入true，主线程跌打处输出cha
43.      此时主线程并没有退出，一直等待关闭channel
44.     */
45. }

```

go锁（和Qt一样）

互斥锁

```

1. var mutex sync.Mutex
2. mutex.Lock()

```

读写锁

```

1. var rwMutex sync.RWMutex
2. rwMutex.Lock()
3. rwMutex.Unlock()

```

Golang Web

以类型形式

```

1. // http
2. package main
3. import (
4.     "net/http"
5. )
6. type SingleHost struct {
7.     handler    http.Handler
8.     allowhost string
9. }
10. func (this *SingleHost) ServeHTTP(w http.ResponseWriter, r *http.Request) {
11.     println(r.Host)
12.     if r.Host == this.allowhost {

```

```

13.         this.handler.ServeHTTP(w, r)
14.     } else {
15.         w.WriteHeader(403)
16.     }
17. }
18. func myHandler(w http.ResponseWriter, r *http.Request) {
19.     w.Write([]byte("hello world"))
20. }
21. func main() {
22.     single := &SingleHost{
23.         handler: http.HandlerFunc(myHandler),
24.         allowhost: "localhost:8080",
25.     }
26.     http.ListenAndServe(":8080", single)
27. }

```

以函数形式

```

1. // http
2. package main
3. import (
4.     "net/http"
5. )
6. func SingleHost(handler http.Handler, allowhost string) http.Handler {
7.     fn := func(w http.ResponseWriter, r *http.Request) {
8.         println(r.Host)
9.         if r.Host == allowhost {
10.             handler.ServeHTTP(w, r)
11.         } else {
12.             w.WriteHeader(403)
13.         }
14.     }
15.     return http.HandlerFunc(fn)
16. }
17. func myHandler(w http.ResponseWriter, r *http.Request) {
18.     w.Write([]byte("hello world"))
19. }
20. func main() {
21.     single := SingleHost(http.HandlerFunc(myHandler), "localhost:8080")
22.     http.ListenAndServe(":8080", single)
23. }

```

对Handler自定义

```

1. // http
2. package main
3. import (
4.     "net/http"
5.     "net/http/httptest"
6. )
7. type ModifierMiddleware struct {
8.     handler http.Handler
9. }
10. func (this *ModifierMiddleware) ServeHTTP(w http.ResponseWriter, r *http.Request) {
11.     //自定义Handler，自定义返回Header和内容
12.     rec := httptest.NewRecorder()
13.     this.handler.ServeHTTP(rec, r)

```

```
14.     for k, v := range rec.Header() {
15.         w.Header()[k] = v
16.     }
17.     w.Header().Set("go-web-foundation", "vip")
18.     w.WriteHeader(418)
19.     w.Write([]byte("hey, this is middleware!"))
20.     w.Write(rec.Body.Bytes())
21. }
22. func myHandler(w http.ResponseWriter, r *http.Request) {
23.     w.Write([]byte("hello world"))
24. }
25. func main() {
26.     single := &ModifierMiddleware{
27.         handler: http.HandlerFunc(myHandler),
28.     }
29.     http.ListenAndServe(":8080", single)
30. }
```

三 In Container

Docker

Docker基础

Docker子命令简单分类：

子命令分类	子命令
Docker环境信息	info、version
容器生命周期管理	create、exec、kill、pause、restart、rm、run、start、stop、unpause
镜像仓库命令	login、logout、pull、push、search
镜像管理	build、images、import、load、rmi、save、tag、commit
容器运维操作	attach、export、inspect、port、ps、rename、stats、top、wait、cp、diff、update
容器资源管理	volume、network
系统日志管理	events、history、logs

比较有意思的几个命令：

(1) 容器从生到死整个生命周期

```
1. root@devstack:/home/sammy# docker create --name web31 training/webapp python app.py #创建
2. 7465f4cb7c49555af32929bd1bc4213f5e72643c0116450e495b71c7ec128502
3. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' web31 #其状态为 cre
4. created
5. root@devstack:/home/sammy# docker start web31 #启动容器
6. web31
7. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' web31 #其状态为 run
8. running
9. root@devstack:/home/sammy# docker pause web31 #暂停容器
10. web31
11. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' web31
12. paused
```

```

13. root@devstack:/home/sammy# docker unpause web31 #继续容器
14. web31
15. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' web31
16. running
17. root@devstack:/home/sammy# docker rename web31 newweb31 #重命名
18. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' newweb31
19. running
20. root@devstack:/home/sammy# docker top newweb31 #在容器中运行 top 命令
21.          UID          PID          PPID          C          STIME
22.          root        5009        4979          0          16:28
23. root@devstack:/home/sammy# docker logs newweb31 #获取容器的日志
24. * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
25. root@devstack:/home/sammy# docker stop newweb31 #停止容器
26. newweb31
27. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' newweb31
28. exited
29. root@devstack:/home/sammy# docker rm newweb31 #删除容器
30. newweb31
31. root@devstack:/home/sammy# docker inspect --format='{{.State.Status}}' newweb31
32. Error: No such image, container or task: newweb31

```

(2) docker stop 和 docker kill

docker stop 命令执行的时候，会先向容器中PID为1的进程发送系统信号 SIGTERM，然后等待容器中的应用程序终止执行，如果等待时间达到设定的超时时间（默认为 10秒，用户可以指定特定超时时长），会继续发送 SIGKILL 的系统信号强行kill掉进程。在容器中的应用程序，可以选择忽略和不处理SIGTERM信号，不过一旦达到超时时间，程序就会被系统强行kill掉，因为SIGKILL信号是直接发往系统内核的，应用程序没有机会去处理它。 docker kill 命令会直接发出SIGKILL的系统信号，以强行终止容器中程序的运行。

3) 使用 docker cp 在 host 和 container 之间拷贝文件或者目录

```

1. root@devstack:/home/sammy# docker cp /home/sammy/mydockercfg/Dockerfile web5:/webapp #从 c
2. root@devstack:/home/sammy#
3. root@devstack:/home/sammy# docker cp web5:/webapp/Dockerfile /home/sammy/Dockerfile #从 c
4. root@devstack:/home/sammy# ls /home/sammy
5. chroot devstack Dockerfile mongodocker mydockercfg webapp

```

(4) 可以使用 docker rm \$(docker ps -q -a) 一次性删除所有的容器， docker rmi \$(docker images -q) 一次性删除所有的镜像。

(5) docker export+import和docker save+load区别：

save只能对image用，产生的文件需要用load来生成image； export的对象是container，产生的文件需要用import来生成image。

Export命令用于持久化容器（不是镜像）

Save命令用于持久化镜像（不是容器）

二者区别：

导出（Export）后再导入（Import）（exported-imported）的镜像会丢失所有的历史，而保存（Save）后再加载（Load）（saveed-loaded）的镜像没有丢失历史和层(layer)。这意味着使用导出后再导入的方式，你将无法回滚到之前的层(layer)，同时，使用保存（Save）后再加载（Load）的方式持久化整个镜像，就可以做到层回滚（可以执行docker tag <LAYER ID> <IMAGE NAME>来回滚之前的层）。

镜像(image)是动态的容器的静态表示 (specification)，包括容器所要运行的应用代码以及运行时的配置。

Docker 镜像包括一个或者多个只读层 (read-only layers)，因此，镜像一旦被创建就再也不能被修改了。一个运行着的Docker 容器是一个镜像的实例 (instantiation)。从同一个镜像中运行的容器包含有相同的应用代码和运行时依赖。但是不像镜像是静态的，每个运行着的容器都有一个可写层 (writable layer，也成为容器层

container layer) , 它位于底下的若干只读层之上。运行时的所有变化，包括对数据和文件的写和更新，都会保存在这个层中。因此，从同一个镜像运行的多个容器包含了不同的容器层。

Docker 有两种方式来创建一个容器镜像：

- 创建一个容器，运行若干命令，再使用 docker commit 来生成一个新的镜像。不建议使用这种方案。
- 创建一个 Dockerfile 然后再使用 docker build 来创建一个镜像。大多人会使用 Dockerfile 来创建镜像。

```
1. root@devstack:/home/sammy/ntponubuntu# docker commit -a 'Lynzabo' -m 'Lynzabo nginx' comm
2. 使用 docker history 命令查看该镜像中每一层的信息
3. root@devstack:/home/sammy/ntponubuntu# docker history af678df648bc
4. IMAGE          CREATED      CREATED BY
5. af678df648bc  16 hours ago /bin/sh -c #(nop)  CMD ["/usr/sbin/ntp"]
6. f5c96137bec9  16 hours ago /bin/sh -c #(nop)  EXPOSE 5555/tcp
7. 9cc05cf6f48d  16 hours ago /bin/sh -c apt-get -y install ntp
8. 694a19d54103  16 hours ago /bin/sh -c apt-get update
9. c4299e3f774c  17 hours ago /bin/sh -c #(nop)  MAINTAINER sammy "sammy@sa
10. 4a725d3b3b1c  3 weeks ago /bin/sh -c #(nop)  CMD ["/bin/bash"]
11. <missing>      3 weeks ago /bin/sh -c mkdir -p /run/systemd && echo 'doc
12. <missing>      3 weeks ago /bin/sh -c sed -i 's/^#\s*\(\deb.*universe\)\$/'
13. <missing>      3 weeks ago /bin/sh -c rm -rf /var/lib/apt/lists/*
14. <missing>      3 weeks ago /bin/sh -c set -xe  && echo '#!/bin/sh' > /u
15. <missing>      3 weeks ago /bin/sh -c #(nop) ADD file:ada91758a31d8de3c7
```

```
1. docker build -t ubuntu/dockerfile-user-1000 .
```

CMD和ENTRYPOINT：在容器被创建后执行的命令（启动容器时执行的命令），和 RUN 不同，它是在构造容器时候所执行的命令。

CMD和ENTRYPOINT区别：

这两个指令都指定了运行容器时所运行的命令。以下是它们共存的一些规则：

- 一个Dockerfile里只能有一个CMD，如果有多个，只有最后一个生效。
- CMD 可以用来指定 ENTRYPOINT 指令的参数

同时有 CMD 和 ENTRYPOINT

```
1. FROM ubuntu:14.04
2. MAINTAINER Sammy Liu <sammy.liu@unknow.com>
3. CMD top
4. ENTRYPOINT ps
```

```
1. 此时会运行的指令为 /bin/sh -c ps /bin/sh -c top
```

CMD 作为 ENTRYPOINT 的参数

```
1. FROM ubuntu:14.04
2. MAINTAINER Sammy Liu <sammy.liu@unknow.com>
3. CMD ["-n", "10"]
4. ENTRYPOINT top
```

启动容器后运行的命令为 /bin/sh -c top -n 10。

Dockerfile的EXPOSE或者--expose只是为其他命令提供所需信息的元数据，或者只是告诉容器操作人员有哪些已知选择。它只是作为记录机制，也就是告诉用户哪些端口会提供服务。它保存在容器的元数据中。

使用 -p 发布特定端口。如果该端口已经被 exposed，则发布它；如果它还没有被 exposed，则它会被 exposed 和 published。Docker 不会检查容器端口的正确性。-p 16060:8080

使用 -P 时 Docker 会自动将所有已经被 exposed 的端口映射到主机随机端口。

namespace

Docker 容器本质上是宿主机上的进程。Docker 通过 namespace 实现了资源隔离，通过 cgroups 实现了资源限制，通过写时复制机制（copy-on-write）实现了高效的文件操作。

namespace 包括 6 种 namespace 支持：Mount namespace、UTS namespace、IPC namespace、PID namespace、Network namespace、docker 1.10 支持了 User namespace。

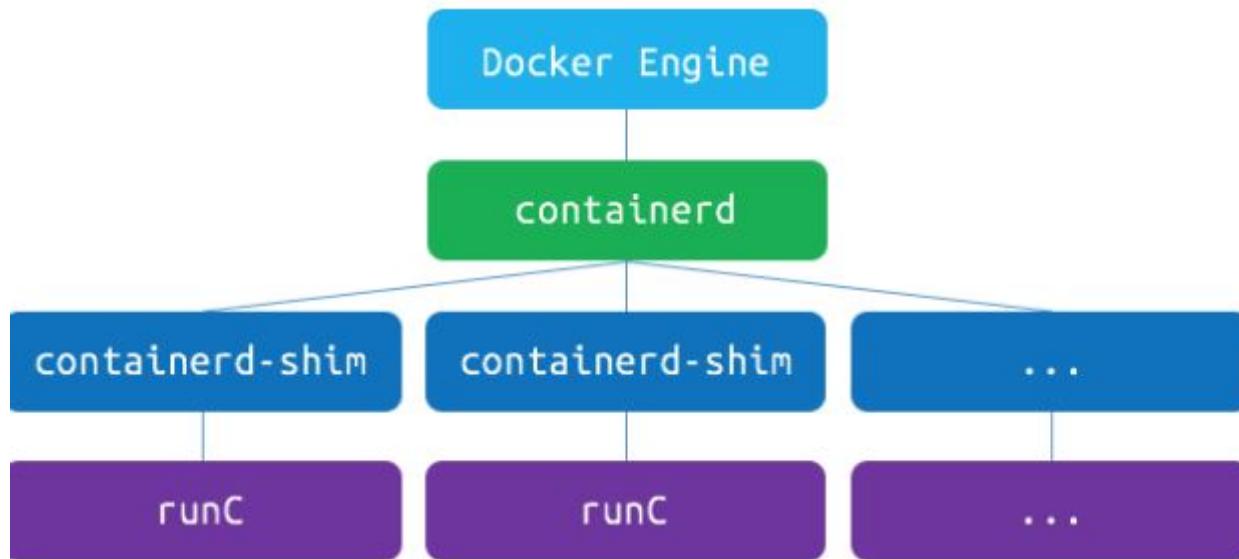
可以在 /proc/[pid]/ns 目录下，看到当前进程的 6 大 namespace。

- UTS namespace，提供了主机名和域名的隔离，使得容器拥有独立的主机名和域名，在网络上可视作一个独立的节点。
- IPC namespace，设计常见的信号量、消息队列和共享内存。同一个 IPC namespace 下的进程彼此可见，不同 IPC namespace 下的进程则互相不可见。
- PID namespace 隔离是对进程 PID 重新标号，即两个不同 namespace 下的进程可以有相同的 PID。每个 PID namespace 中的第一个进程“PID 1”，都会像传统 Linux 中的 init 进程一样拥有特权，起特殊作用。

我们能看到同一个进程，在容器内外的 PID 是不同的：

- 在容器内 PID 是 1，PPID 是 0。
- 在容器外 PID 是 10817，PPID 是 10809 即 docker-containerd-shim 进程。

关于 containerd、containerd-shim 和 container 的关系，如下图说明（后面详讲）：



- Docker Engine 管理着镜像，然后移交给 containerd 运行，containerd 再使用 runC 运行容器。
- Containerd 是一个简单的守护进程，它可以使用 runC 管理容器，使用 gRPC 暴露容器的其他功能。它管理容器的开始、停止、暂停和销毁。由于容器运行时是孤立的引擎，引擎最终能够启动和升级而无需重新启动容器。
- runC 是一个轻量级的工具，它是用来运行容器的，只用来做这一件事，并且这一件事要做好。runC 基本上是一个小命令行工具且它可以不用通过 Docker 引擎，直接就可以使用容器。

因此，容器中的主应用在 host 上的父进程是 containerd-shim，是它通过工具 runC 来启动这些进程的。

这也能够看出来，pid namespace 通过将 host 上 PID（不是 PPID 哈）映射为容器内的 PID，使得容器内的进程看起来有个独立的 PID 空间。

- mount namespace 通过隔离文件系统挂载点对隔离文件系统提供支持，不同 mount namespace 中的文件结构发生变化也互不影响。

- network namespace主要提供了关于网络资源的隔离，包括网络设备、IPv4和IPv6协议栈、IP路由表、防火墙、/proc/net目录、/sys/class/net目录、套接字（socket）等。

容器里通过veth设备对实现不同网络命名空间间通信。

默认情况下，当 docker 实例被创建出来后，使用 ip netns 命令无法看到容器实例对应的 network namespace。这是因为 ip netns 命令是从 /var/run/netns 文件夹中读取内容的。

```

1. 步骤：
2. 1. 找到容器的主进程 ID
3. root@devstack:/home/sammy# docker inspect --format '{{.State.Pid}}' web5
4. 2704
5. 2. 创建 /var/run/netns 目录以及符号连接
6. root@devstack:/home/sammy# mkdir /var/run/netns
7. root@devstack:/home/sammy# ln -s /proc/2704/ns/net /var/run/netns/web5
8. 3. 此时可以使用 ip netns 命令了
9. root@devstack:/home/sammy# ip netns
10. web5
11. root@devstack:/home/sammy# ip netns exec web5 ip addr
12. 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
13.   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
14.   inet 127.0.0.1/8 scope host lo
15.     valid_lft forever preferred_lft forever
16.   inet6 ::1/128 scope host
17.     valid_lft forever preferred_lft forever
18. 15: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
19.   link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
20.   inet 172.17.0.3/16 scope global eth0
21.     valid_lft forever preferred_lft forever
22.     inet6 fe80::42:acff:fe11:3/64 scope link
23.       valid_lft forever preferred_lft forever

```

- user namespace主要隔离了安全相关的标识符（identifier）和属性（attribute），包括用户ID、用户组ID、root目录、key（指密钥）以及特殊权限。

一个普通用户的进程通过clone()创建的新进程在新user namespace中可以拥有不同的用户和用户组。这意味着一个进程在容器外属于一个没有特权的普通用户，但是它创建的容器进程却属于所有权限的超级用户。

在 Docker 1.10 版本之前，Docker 是不支持 user namespace。也就是说，默认地，容器内的进程的运行用户就是 host 上的 root 用户，这样的话，当 host 上的文件或者目录作为 volume 被映射到容器以后，容器内的进程其实是有 root 的几乎所有权限去修改这些 host 上的目录的，这会有很大的安全问题。

举例：

启动一个容器：

```
1. docker run -d -v /bin:/host/bin --name web34 training/webapp python app.py
```

此时进程的用户在容器内和外都是root，它在容器内可以对 host 上的 /bin 目录做任意修改。

Docker在1.10版本中对user namespace进行了支持。只要用户在启动Docker daemon的时候指定了--userns-remap，那么当用户运行容器时，容器内部的root用户并不等于宿主机内的root用户，而是映射到宿主机上的普通用户。

cgroups

cgroups的实现本质上是给任务挂上钩子，当任务运行的过程中涉及某种资源时，就会触发钩子上所附带的子系统进行检测，根据资源类别的不同，使用对应的技术进行资源限制和优先级分配。

Linux中cgroup的实现形式表现为一个文件系统，因此需要mount这个文件系统才能够使用（也有可能已经mount好了），挂载成功后，就能看到各类子系统。

在/sys/fs/cgroup路径。

```
1. [root@localhost ~]# mount -t cgroup
2. cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,relea
3. cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
4. cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpuacct)
5. cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,ne
6. cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
7. cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_eve
8. cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
9. cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
10. cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
11. cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
12. cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
13. [root@localhost ~]#
```

```
1. [root@localhost cgroup]# ls /sys/fs/cgroup/ -l
2. total 0
3. drwxr-xr-x. 5 root root 0 Jun 8 02:26 blkio
4. lrwxrwxrwx. 1 root root 11 Jun 8 02:26 cpu -> cpu,cpuacct
5. lrwxrwxrwx. 1 root root 11 Jun 8 02:26 cpuacct -> cpu,cpuacct
6. drwxr-xr-x. 6 root root 0 Jun 8 02:26 cpu,cpuacct
7. drwxr-xr-x. 3 root root 0 Jun 8 02:26 cpuset
8. drwxr-xr-x. 5 root root 0 Jun 8 02:26 devices
9. drwxr-xr-x. 3 root root 0 Jun 8 02:26 freezer
10. drwxr-xr-x. 3 root root 0 Jun 8 02:26 hugetlb
11. drwxr-xr-x. 5 root root 0 Jun 8 02:26 memory
12. lrwxrwxrwx. 1 root root 16 Jun 8 02:26 net_cls -> net_cls,net_prio
13. drwxr-xr-x. 3 root root 0 Jun 8 02:26 net_cls,net_prio
14. lrwxrwxrwx. 1 root root 16 Jun 8 02:26 net_prio -> net_cls,net_prio
15. drwxr-xr-x. 3 root root 0 Jun 8 02:26 perf_event
16. drwxr-xr-x. 3 root root 0 Jun 8 02:26 pids
17. drwxr-xr-x. 5 root root 0 Jun 8 02:26 systemd
18. [root@localhost cgroup]#
```

如在/sys/fs/cgroup的cpu子目录下创建控制组，控制组目录创建后，它下面自动回有如下文件。

```
1. [root@localhost cpu]# ls
2. cgroup.clone_children  cpuacct.stat          cpu.cfs_quota_us   cpu.stat
3. cgroup.event_control   cpuacct.usage         cpu.rt_period_us  notify_on_release
4. cgroup.procs           cpuacct.usage_percpu  cpu.rt_runtime_us release_agent
5. cgroup.sane_behavior   cpu.cfs_period_us    cpu.shares        tasks
6. [root@localhost cpu]# mkdir cg1
7. [root@localhost cpu]# ls cg1
8. cgroup.clone_children  cpuacct.stat          cpu.cfs_period_us  cpu.rt_runtime_us  notify
9. cgroup.event_control   cpuacct.usage         cpu.cfs_quota_us  cpu.shares        tasks
10. cgroup.procs           cpuacct.usage_percpu  cpu.rt_period_us   cpu.stat
11. [root@localhost cpu]#
```

下面展示了如何限制PID为11292的进程的CPU使用配额：

hello.c

```
1. int main(void)
2. {
3.     int i = 0;
4.     for(;;) i++;
5.     return 0;
6. }
```

编译并运行：

```
1. [root@localhost kernel_test]# vim hello.c
2. [root@localhost kernel_test]# gcc hello.c -o hello && ./hello &
[1] 11286
3. [root@localhost kernel_test]# ps -ef|grep hello
4. root    11292 11286 99 02:39 pts/0    00:00:08 ./hello
5. root    11294 11202  0 02:39 pts/0    00:00:00 grep --color=auto hello
6. [root@localhost kernel_test]#
7. [root@localhost kernel_test]#
```

查看PID为11292该进程占用CPU，看到占用CPU一直是99%以上。

```
1. [root@localhost kernel_test]# top
2. top - 02:42:08 up 15 min, 1 user, load average: 0.96, 0.41, 0.19
3. Tasks: 87 total, 2 running, 85 sleeping, 0 stopped, 0 zombie
4. %Cpu(s): 99.5 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.5 si, 0.0 st
5. KiB Mem : 500380 total, 153324 free, 73464 used, 273592 buff/cache
6. KiB Swap: 1572860 total, 1572860 free, 0 used. 386328 avail Mem
7. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8. 11292 root 20 0 4156 340 272 R 99.7 0.1 2:18.54 hello
9. 388 root 20 0 0 0 0 S 0.3 0.0 0:00.14 xfsaild/dm-0
10. 1 root 20 0 128100 6724 3968 S 0.0 1.3 0:00.97 systemd
11. 2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
12. 3 root 20 0 0 0 0 S 0.0 0.0 0:00.03 ksoftirqd/0
13. 6 root 20 0 0 0 0 S 0.0 0.0 0:00.03 kworker/u2:0
14. 37 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kworker/u2:1
```

下面限制PID为11292这个进程的CPU

```
1. [root@localhost kernel_test]# cd /sys/fs/cgroup/cpu/cg1/
2. # 限制11292进程
3. [root@localhost cg1]# echo 11292 >> /sys/fs/cgroup/cpu/cg1/tasks
4. # 将CPU限制为最高使用20%
5. [root@localhost cg1]# echo 20000 > /sys/fs/cgroup/cpu/cg1/cpu.cfs_quota_us
6. [root@localhost cg1]#
```

现在查看CPU占用,确实最多20%。

```
1. [root@localhost cg1]# top
2. top - 02:45:22 up 19 min, 1 user, load average: 0.50, 0.56, 0.30
3. Tasks: 88 total, 2 running, 86 sleeping, 0 stopped, 0 zombie
4. %Cpu(s): 14.3 us, 0.0 sy, 0.0 ni, 85.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
5. KiB Mem : 500380 total, 153204 free, 73584 used, 273592 buff/cache
6. KiB Swap: 1572860 total, 1572860 free, 0 used. 386208 avail Mem
7. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8. 11292 root 20 0 4156 340 272 R 20.0 0.1 4:33.58 hello
9. 11244 root 20 0 0 0 0 S 0.3 0.0 0:00.13 kworker/0:0
10. 1 root 20 0 128100 6724 3968 S 0.0 1.3 0:00.97 systemd
11. 2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
12. 3 root 20 0 0 0 0 S 0.0 0.0 0:00.03 ksoftirqd/0
13. 6 root 20 0 0 0 0 S 0.0 0.0 0:00.03 kworker/u2:0
14. 36 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kthrotld
```

在Docker的实现中，Docker daemon会在单独挂载了每一个子系统的控制组目录（比如/sys/fs/cgroup/cpu）下创建一个名为docker的控制组，然后在docker控制组里面，再为每个容器创建一个以容器ID为名称的容器控制组，这个容器里的所有进程的进程号都会写到该控制组tasks中，并且在控制文件（比如cpu.cfs_quota_us）中写入预设的限制参数值。

如CPU控制组下docker控制组的层级结构如下：

```
$ tree cgroup/cpu/docker
cgroup/cpu/docker/
└── 0e8220dbfeac5cb96eb34c4b1a0d648e9358f205bec8c803bc1f7fc178cb8f78 #这是容器ID
    ├── cgroup.clone_children
    ├── cgroup.procs
    ├── cpu.cfs_period_us
    ├── cpu.cfs_quota_us #这里有值
    ├── cpu.rt_period_us
    └── cpu.rt_runtime_us

    ├── cpu.shares
    ├── cpu.stat
    ├── notify_on_release
    └── tasks

    ├── cgroup.clone_children
    ├── cgroup.procs
    ├── cpu.cfs_period_us
    ├── cpu.cfs_quota_us
    ├── cpu.rt_period_us
    ├── cpu.rt_runtime_us
    ├── cpu.shares
    ├── cpu.stat
    ├── notify_on_release
    └── tasks
```

Docker 会将容器中的进程的 ID 加入到各个资源对应的 tasks 文件中。

Docker支持的容器资源配置（重点,Lynzabo）

CPU

- CPU份额控制：

docker提供了–cpu-shares参数，在创建容器时指定容器所使用的CPU份额值。

cpu-shares的值不能保证可以获得1个cpu或者多少GHz的CPU资源，仅仅只是一个弹性的加权值。默认情况下，每个docker容器的cpu份额都是1024。单独一个容器的份额是没有意义的，只有在同时运行多个容器时，容器的

cpu加权的效果才能体现出来。例如，两个容器A、B的cpu份额分别为1000和500，在cpu进行时间片分配的时候，容器A比容器B多一倍的机会获得CPU的时间片，但分配的结果取决于当时主机和其他容器的运行状态，实际上也无法保证容器A一定能获得CPU时间片。比如容器A的进程一直是空闲的，那么容器B是可以获取比容器A更多的CPU时间片的。极端情况下，比如说主机上只运行了一个容器，即使它的cpu份额只有50，它也可以独占整个主机的cpu资源。

cgroups只在容器分配的资源紧缺时，也就是说在需要对容器使用的资源进行限制时，才会生效。因此，无法单纯根据某个容器的cpu份额来确定有多少cpu资源分配给它，资源分配结果取决于同时运行的其他容器的cpu分配和容器中进程运行情况。

使用示例：

使用下面命令创建容器：

```
1. docker run -tid --cpu-shares 100 ubuntu:stress
```

则最终生成的cgroup的cpu份额配置可以下面的文件中找到：

```
1. root@ubuntu:~# cat /sys/fs/cgroup/cpu/docker/<容器的完整长ID>/cpu.shares
2. 100
```

- CPU周期控制

docker提供了--cpu-period、--cpu-quota两个参数控制容器可以分配到的CPU时钟周期。--cpu-period是用来指定容器对CPU的使用要在多长时间内做一次重新分配，而--cpu-quota是用来指定在这个周期内，最多可以有多少时间用来跑这个容器。跟--cpu-shares不同的是这种配置是指定一个绝对值，而且没有弹性在里面，容器对CPU资源的使用绝对不会超过配置的值。

--cpu-period和--cpu-quota的单位为微秒（ μs ）。--cpu-period的最小值为1000微秒，最大值为1秒（ $10^6 \mu\text{s}$ ），默认值为0.1秒（100000 μs ）。--cpu-quota的值默认为-1，表示不做控制。

举个例子，如果容器进程需要每1秒使用单个CPU的0.2秒时间，可以将--cpu-period设置为1000000（即1秒），--cpu-quota设置为200000（0.2秒）。当然，在多核情况下，如果允许容器进程需要完全占用两个CPU，则可以将--cpu-period设置为100000（即0.1秒），--cpu-quota设置为200000（0.2秒）。

使用示例：

使用下面命令创建容器，

```
1. docker run -tid --cpu-period 100000 --cpu-quota 200000 ubuntu
```

则最终生成的cgroup的cpu周期配置可以下面的文件中找到：

```
1. root@ubuntu:~# cat /sys/fs/cgroup/cpu/docker/<容器的完整长ID>/cpu.cfs_period_us
2. 100000
3. root@ubuntu:~# cat /sys/fs/cgroup/cpu/docker/<容器的完整长ID>/cpu.cfs_quota_us
4. 200000
```

- CPU core控制

对多核CPU的服务器，docker还可以控制容器运行限定使用哪些cpu内核和内存节点，即使用--cpuset-cpus和--cpuset-mems参数。如果服务器只有一个内存节点，则--cpuset-mems的配置基本上不会有明显效果。

使用示例：

```
1. docker run -tid --name cpu1 --cpuset-cpus 0-2 ubuntu
```

表示创建的容器只能用0、1、2这三个内核。最终生成的cgroup的cpu内核配置如下：

```
1. root@ubuntu:~# cat /sys/fs/cgroup/cpuset/docker/<容器的完整长ID>/cpuset.cpus
2. 0-2
```

通过

```
1. docker exec <容器ID> taskset -c -p 1
```

(容器内部第一个进程编号一般为1), 可以看到容器中进程与CPU内核的绑定关系, 可以认为达到了绑定CPU内核的目的。

上面可以混合着使用。

- cpu-shares控制只发生在容器竞争同一个内核的时间片时。
- cpuset-cpus指定容器A使用内核0, 容器B只使用内核1, 在主机上只有这两个容器使用对应内核的情况下, 它们各自占用全部的内核资源。
- cpu-period、cpu-quota这两个参数一般联合使用, 在单核情况或者通过cpuset-cpus强制容器使用一个cpu内核的情况下, 当cpu-quota超过cpu-period, 也不会使容器使用更多的CPU资源。

下面命令创建测试:

```
1. docker run -tid --name cpu2 --cpuset-cpus 3 --cpu-shares 512 ubuntu:stress stress -c 10
2. docker run -tid --name cpu3 --cpuset-cpus 3 --cpu-shares 1024 ubuntu:stress stress -c 10
```

上面的ubuntu:stress镜像安装了stress工具来测试CPU和内存的负载。两个容器的命令stress -c 10&, 这个命令将会给系统一个随机负载, 产生10个进程, 每个进程都反复不停的计算由rand()产生随机数的平方根, 直到资源耗尽。

观察到宿主机上的CPU试用率如下图所示, 第三个内核的使用率接近100%, 并且一批进程的CPU使用率明显存在2:1的比例的对比:

```
top - 18:19:51 up 8:42, 1 user, load average: 15.53, 18.47, 17.14
Tasks: 182 total, 22 running, 159 sleeping, 0 stopped, 1 zombie
%Cpu0 : 1.0 us, 1.3 sy, 0.0 ni, 96.0 id, 1.7 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 1.3 us, 0.7 sy, 0.0 ni, 98.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 0.3 us, 2.0 sy, 0.0 ni, 97.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 99.0 us, 1.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 4048076 total, 2125060 used, 1923016 free, 202076 buffers
KiB Swap: 4194300 total, 0 used, 4194300 free. 804284 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
16398	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16399	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16400	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16401	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16402	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16403	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16404	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16405	root	20	0	7308	96	0	R	6.6	0.0	0:01.29	stress
16406	root	20	0	7308	96	0	R	6.6	0.0	0:01.28	stress
16407	root	20	0	7308	96	0	R	6.3	0.0	0:01.28	stress
16323	root	20	0	7308	100	0	R	3.7	0.0	0:02.06	stress
16315	root	20	0	7308	100	0	R	3.3	0.0	0:02.05	stress
16317	root	20	0	7308	100	0	R	3.3	0.0	0:02.05	stress
16318	root	20	0	7308	100	0	R	3.3	0.0	0:02.06	stress
16321	root	20	0	7308	100	0	R	3.3	0.0	0:02.05	stress
16322	root	20	0	7308	100	0	R	3.3	0.0	0:02.06	stress
16324	root	20	0	7308	100	0	R	3.3	0.0	0:02.06	stress
16316	root	20	0	7308	100	0	R	3.0	0.0	0:02.05	stress
16319	root	20	0	7308	100	0	R	3.0	0.0	0:02.05	stress
16320	root	20	0	7308	100	0	R	3.0	0.0	0:02.05	stress

容器cpu2的CPU使用如下所示：

```
top - 10:25:35 up 8:48, 0 users, load average: 20.37, 19.75, 18.13
Tasks: 13 total, 11 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.7 us, 1.0 sy, 0.0 ni, 93.4 id, 5.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.7 us, 0.3 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 1.7 us, 0.7 sy, 0.0 ni, 95.0 id, 2.7 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 99.7 us, 0.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 4048076 total, 2131064 used, 1917012 free, 202084 buffers
KiB Swap: 4194300 total, 0 used, 4194300 free. 805384 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
15	root	20	0	7308	100	0	R	3.7	0.0	0:13.52	stress
6	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
7	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
8	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
9	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
11	root	20	0	7308	100	0	R	3.3	0.0	0:13.52	stress
12	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
13	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
14	root	20	0	7308	100	0	R	3.3	0.0	0:13.51	stress
10	root	20	0	7308	100	0	R	3.0	0.0	0:13.51	stress

容器cpu3的CPU使用如下所示：

```
top - 10:28:05 up 8:50, 0 users, load average: 20.11, 19.89, 18.42
Tasks: 13 total, 11 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu0 : 1.2 us, 0.8 sy, 0.0 ni, 96.7 id, 1.2 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 1.2 us, 0.4 sy, 0.0 ni, 95.5 id, 2.9 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 0.8 us, 0.4 sy, 0.0 ni, 98.3 id, 0.4 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 4048076 total, 2129512 used, 1918564 free, 202088 buffers
KiB Swap: 4194300 total, 0 used, 4194300 free. 805648 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
7	root	20	0	7308	96	0	R	7.0	0.0	0:34.19	stress
8	root	20	0	7308	96	0	R	7.0	0.0	0:34.19	stress
10	root	20	0	7308	96	0	R	7.0	0.0	0:34.19	stress
11	root	20	0	7308	96	0	R	7.0	0.0	0:34.19	stress
5	root	20	0	7308	96	0	R	6.6	0.0	0:34.18	stress
6	root	20	0	7308	96	0	R	6.6	0.0	0:34.19	stress
9	root	20	0	7308	96	0	R	6.6	0.0	0:34.19	stress
12	root	20	0	7308	96	0	R	6.6	0.0	0:34.19	stress
13	root	20	0	7308	96	0	R	6.6	0.0	0:34.18	stress
14	root	20	0	7308	96	0	R	6.6	0.0	0:34.18	stress
1	root	20	0	7308	632	536	S	0.0	0.0	0:00.06	stress

Memory

docker也提供了若干参数来控制容器的内存使用配额，可以控制容器的swap大小、可用内存大小等各种内存方面的控制。主要有以下参数：

- **-memory:**设置容器使用的最大内存上限。默认单位为byte，可以使用K、G、M等带单位的字符串。
- **-memory-reservation:**启用弹性的内存共享，当宿主机资源充足时，允许容器尽量多地使用内存，当检测到内存竞争或者低内存时，强制将容器的内存降低到memory-reservation所指定的内存大小。按照官方说法，不设置此选项时，有可能出现某些容器长时间占用大量内存，导致性能上的损失。
- **-memory-swap:**等于内存和swap分区大小的总和，设置为-1时，表示swap分区的大小是无限的。默认单位为byte，可以使用K、G、M等带单位的字符串。

默认情况下，容器可以使用主机上的所有空闲内存。

与CPU的cgroups配置类似，docker会自动为容器在目录/sys/fs/cgroup/memory/docker/<容器的完整长ID>中创建相应cgroupp配置文件。

设置容器的内存上限，参考命令如下所示：

```
1. docker run -tid --name mem1 --memory 128m ubuntu:stress /bin/bash
```

默认情况下，除了`--memory`指定的内存大小以外，`docker`还为容器分配了同样大小的`swap`分区，也就是说，上面的命令创建出的容器实际上最多可以使用256MB内存，而不是128MB内存。如果需要自定义`swap`分区大小，则可以通过联合使用`--memory--swap`参数来实现控制。

对上面的命令创建的容器，可以查看到在`cgroups`的配置文件中，查看到容器的内存大小为128MB ($128 \times 1024 \times 1024 = 134217728\text{B}$)，内存和`swap`加起来大小为256MB ($256 \times 1024 \times 1024 = 268435456\text{B}$)。

```
1. cat /sys/fs/cgroup/memory/docker/<容器的完整ID>/memory.limit_in_bytes  
2. 134217728  
3. cat /sys/fs/cgroup/memory/docker/<容器的完整ID>/memory.memsw.limit_in_bytes  
4. 268435456
```

磁盘IO

`docker`对磁盘IO的控制主要包括以下参数：

- `--device-read-bps`: 限制此设备上的读速度 (bytes per second)，单位可以是kb、mb或者gb。
- `--device-read-iops`: 通过每秒读IO次数来限制指定设备的读速度。
- `--device-write-bps`: 限制此设备上的写速度 (bytes per second)，单位可以是kb、mb或者gb。
- `--device-write-iops`: 通过每秒写IO次数来限制指定设备的写速度。
- `--blkio-weight`: 容器默认磁盘IO的加权值，有效值范围为10-100。

磁盘IO配额控制示例

要使`--blkio-weight`生效，需要保证IO的调度算法为CFQ。可以使用下面的方式查看：

```
1. root@ubuntu:~# cat /sys/block/sda/queue/scheduler  
2. noop [deadline] cfq
```

使用下面的命令创建两个`--blkio-weight`值不同的容器：

```
1. docker run -ti -rm --blkio-weight 100 ubuntu:stress  
2. docker run -ti -rm --blkio-weight 1000 ubuntu:stress
```

在容器中同时执行下面的`dd`命令，进行测试：

```
1. time dd if=/dev/zero of=test.out bs=1M count=1024 oflag=direct
```

使用下面的命令创建容器，并执行命令验证写速度的限制。

```
1. docker run -tid --name disk1 --device-write-bps /dev/sda:1mb ubuntu:stress
```

通过`dd`来验证写速度，可以看到容器的写磁盘速度被成功地限制到了1MB/s。`device-read-bps`等其他磁盘IO限制参数可以使用类似的方式进行验证。

- 容器空间大小限制

在`docker`使用`devicemapper`作为存储驱动时，默认每个容器和镜像的最大大小为10G。如果需要调整，可以在`daemon`启动参数中，使用`dm.basesize`来指定，但需要注意的是，修改这个值，**不仅仅需要重启`docker daemon`**

服务，还会导致宿主机上的所有本地镜像和容器都被清理掉。

使用aufs或者overlay等其他存储驱动时，没有这个限制。

网络带宽

tc对veth限额实现容器的网络限额控制：

1. 找到容器的vethname

```
1. #!/bin/bash
2. #filename:getveth.sh
3. #author:wade
4. container_name=$1
5. if [ -z $1 ] ; then
6.     echo "Usage: ./getveth.sh container_name"
7.     exit 1
8. fi
9.
10. if [ `docker inspect -f "{{.State.Pid}}" ${container_name}` &>>/dev/null && echo 0 || echo
11. echo "no this container:${container_name}"
12. exit 1
13. fi
14. pid=`docker inspect -f "{{.State.Pid}}" ${container_name}`
15. mkdir -p /var/run/netns
16. ln -sf /proc/$pid/ns/net "/var/run/netns/${container_name}"
17. index=`ip netns exec "${container_name}" ip link show eth0 | head -n1 | sed s/:.*//` 
18. let index=index+1
19. vethname=`ip link show | grep "^${index}:" | sed "s/${index}: .*.*/\1/"` 
20. echo $vethname
21. rm -f "/var/run/netns/${container_name}"
```

2. docker容器下载流量限额

在node节点上用tc设置对应veth的流量带宽

```
1. tc qdisc add dev vethname root tbf rate 10mbit latency 50ms burst 10000 mpu 64 mtu 15000
```

3. docker容器上传流量限额

在容器内使用tc设置eth0设备的流量带宽

```
1. tc qdisc add dev eth0 root tbf rate 10mbit latency 50ms burst 10000 mpu 64 mtu 15000
```

Docker架构

Docker使用了传统的client-server架构模式，总架构图如下，用户通过Docker client与Docker daemon建立通信，并将请求发送给后者。而Docker的后者是松耦合结构，不同模块各司其职，有机组合，完成用户的请求。

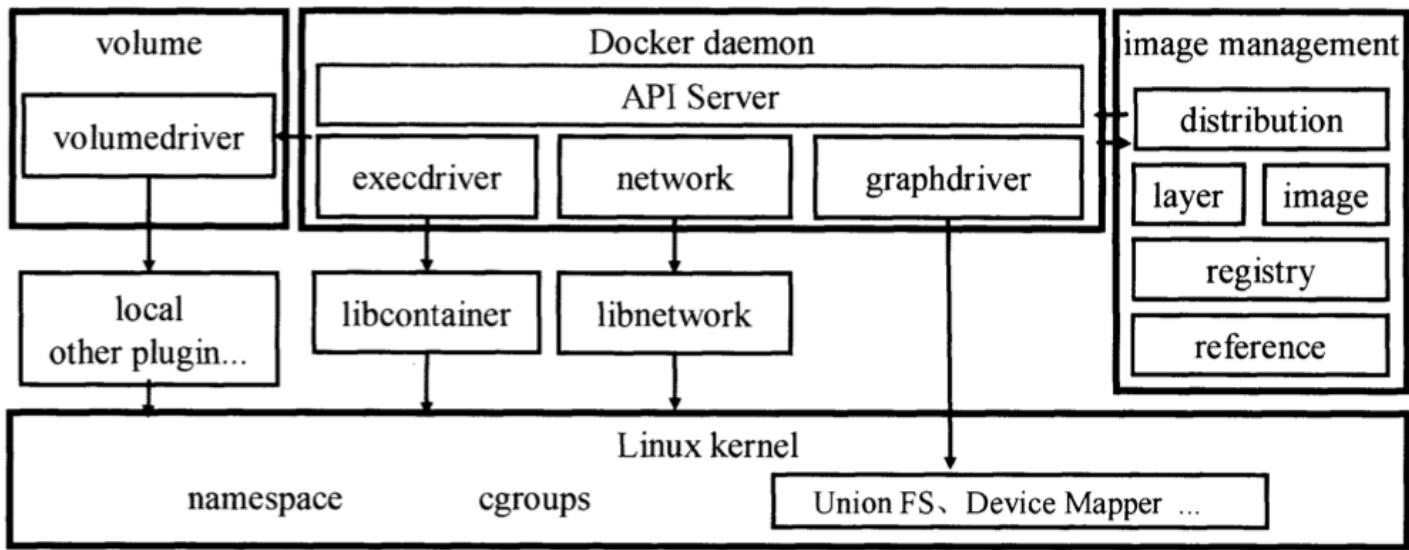


图3-8 Docker架构总览

Docker daemon是Docker架构中的主要用户接口。首先，它提供了API Server用于接收来自Docker client的请求，其后根据不同的请求分发给Docker daemon的不同模块执行相应的工作，其中对容器运行时、volume、镜像以及网络方面的具体实现已经放在daemon以外的模块或项目中。Docker一直致力于将其自己进一步解耦，削减Docker daemon的功能，这是docker新版本的变化。

Docker通过driver模块来实现对Docker容器执行环境的创建和管理。

- 当需要创建Docker容器时，可通过镜像管理（image management）部分的distribution和registry模块从Docker registry中下载镜像，并通过镜像管理的image、reference和layer存储镜像的元数据，通过镜像存储驱动graphdriver将镜像文件存储于具体的文件系统中；
- 当需要为Docker容器创建网络环境时，通过网络管理模块network调用libnetwork创建并配置Docker容器的网络环境；
- 当需要为容器创建数据卷volume时，则通过volume模块调用某个具体的volumedriver，来创建一个数据卷并负责后续的挂载操作；
- 当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。libcontainer是对cgroups和namespace的二次封装，execdriver是通过libcontainer来实现对容器的具体管理，包括利用UTS、IPC、PID、Network、Mount、User等namespace实现容器之间的资源隔离和利用cgroup实现对容器的资源隔离。

当运行容器的命令执行完毕后，一个实际的容器就处于运行状态，该容器拥有独立的文件系统、相对安全且相互隔离的运行环境。

详细看看上面各个模块：

- Docker daemon** Docker daemon是Docker最核心后台进程，负责响应来自Docker client的请求，然后将这些请求翻译成系统调用完成容器管理操作。该进程会在后台启动一个AP Server，负责接收由Docker client发送的请求；接收到的请求将通过Docker daemon分发调度，再由具体的函数来执行请求。
- Docker client** 用来向Docker daemon发起请求，有很多种类，java、go等语言的都有，甚至还有Angular库编写的WebUI格式的客户端。
- 镜像管理** Docker通过distribution、registry、layer、image、reference等模块实现了Docker镜像的管理，我们将这些模块同城为镜像管理（image management）。在Docker 1.10以前的版本中，这一功能是通过graph组件来完成的。下面进行简单介绍，具体细节在镜像管理会详细说明。
 - distribution负责与Docker registry交互，上传下载镜像以及存储与v2 registry有关的元数据。
 - registry模块负责与Docker registry有关的身份验证、镜像查找、镜像验证以及管理registry mirror等交互操作。

- image模块负责与镜像元数据有关的存储、查找，镜像层的索引、查找以及镜像tar包有关的导入、导出等操作。
- reference负责存储本地所有镜像的repository和tag名，并维护与镜像ID之间的映射关系。
- layer模块负责与镜像层和容器层元数据有关的增删改查，并负责将镜像层的增删查改操作映射到实际存储镜像层文件系统的grapdriver模块。
- execdriver、volumedriver、grapdriver Docker daemon负责将用户请求转移成系统调用，进而创建和管理容器。而在具体实现过程中，为了将这些系统调用抽象成为统一的操作接口方便调用者使用，Docker把这些操作分成了容器执行驱动execdriver、volume存储驱动volumedriver、镜像存储驱动graphdriver 3种。
 - execdriver是对linux操作系统的namespaces、cgroups、apparmor、SELinux等容器运行所需的系统操作进行的一层二次封装，其本质作用类似于LXC，但是功能要更全面。这也就是为什么LXC会作为execdriver的一种实现而存在。当然，execdriver最主要的实现，也是现在的默认实现，是Docker官方编写的libcontainer库。
 - volumedriver是volume数据卷存储操作的最终执行者，负责volume的增删改查，屏蔽不同驱动实现的区别，为上层调用者提供一个统一的接口。Docker中作为默认实现的volumedriver是local，默认将文件存储于Docker根目录下的volume文件夹里。其他的volumedriver均是通过外部插件实现的。
 - graphdriver是所有与容器镜像相关操作的最终执行者。graphdriver会在Docker工作目录下维护一组与镜像层对应的目录，并记下镜像层之间的关系以及与具体的graphdriver实现相关的元数据。这样，用户对镜像的操作最终会被映射成对这些目录文件以及元数据的增删改查，从而屏蔽掉不同文件存储实现对于上层调用者的影响。在linux环境下，目前Docker已经支持的graphdriver包括aufs、btrfs、zfs、devicemapper、overlay和vfs。
- network 在Docker1.9版本以前，网络是通过networkdriver模块以及libcontainer库完成的，现在这部分功能已经分离成一个libnetwork库独立维护了。libnetwork抽象出了一个容器网络模型（Container Network Model, CNM），并给调用者提供了一个统一抽象接口，其目标并不仅限于Docker容器。CNM模型对真实的容器网络抽象出了沙盒（sandbox）、端点（endpoint）、网络（network）这3种对象，由具体网络驱动（包括内置的Bridge、Host、None和overlay驱动以及通过插件配置的外部驱动）操作对象，并通过网络控制其提供一个统一接口供调用者管理网络。网络驱动负责实现具体的操作，包括创建容器网络所需的网络，容器的network namespace，这个网络所需的虚拟网卡，分配通信所需的IP，服务访问的端口和容器与宿主机之间的端口映射，设置hosts、resolv.conf、iptables等。

镜像、私有仓库

Docker镜像是一个只读的Docker容器模板，含有启动Docker容器所需的文件系统结构以及内容：

- 分层：每一个镜像都由一系列的“镜像层”组成。
- 写时复制：在多个容器之间共享镜像，每个容器在启动的时候并不需要单独复制一份镜像文件，而是将所有镜像层以只读的方式挂载到一个挂载点，再在上面覆盖一个可读写的容器层。再为更改文件内容时，所有容器共享同一份数据，只有在Docker容器运行过程中文件系统发生变化时，才会把变化的文件内容写到可读写层，并隐藏只读层中的老版本文件。比如基于一个image启动多个Container，如果为每个Container都去分配一个image一样的文件系统，那么将会占用大量的磁盘空间。而写时复制技术可以让所有的容器共享image的文件系统，所有数据都从image中读取，只有当要对文件进行写操作时，才从image里把要写的文件复制到自己的文件系统进行修改。所以无论有多少个容器共享同一个image，所做的写操作都是对从image中复制到自己的文件系统中的复本上进行，并不会修改image的源文件，且多个容器操作同一个文件，会在每个容器的文件系统里生成一个复本，每个容器修改的都是自己的复本，相互隔离，相互不影响。使用写时复制可以有效的提高磁盘的利用率。
- 用时分配：启动一个容器，并不会为这个容器预分配一些磁盘空间，而是当有新文件写入时，才按需分配新空间。

- 内容寻址：之前是每个镜像层都随机生成一个UUID，新模型是对镜像层的内容计算，是把data的内容经过sha256做hash得出的结果。

私有仓库

docker registry v1和v2区别：

v1目录，镜像仓库下存储结构：

```
root@9aa956573328:/tmp/registry# ls -R
.:
images repositories

./images:
7322fbe74aa5632b33a400959867c8ac4290e9c5112877a7754be70cfe5d66e9  f1b10cd8424
c852f6d61e65cddf1e8af1f6cd7db78543bfb83cdcd36845541cf6d9dfef20a0

./images/7322fbe74aa5632b33a400959867c8ac4290e9c5112877a7754be70cfe5d66e9:
_checksum ancestry json layer

./images/c852f6d61e65cddf1e8af1f6cd7db78543bfb83cdcd36845541cf6d9dfef20a0:
_checksum ancestry json layer

./images/f1b10cd842498c23d206ee0cbeaa9de8d2ae09ff3c7af2723a9e337a6965d639:
_checksum ancestry json layer

./repositories:
library

./repositories/library:
centos

./repositories/library/centos:
_index_images json tag_latest taglatest_json
```

最上面是两层结构images和repositories，关注一下images里面的内容，长字符串的是镜像ID，最叶子节点有一个layer和Ancestry。layer就是这一层文件系统的tar包，Ancestry中存储的是它父亲层ID。当使用pull的时候，就是根据链表这种结构，根据子节点，找到父节点。只能单线程的串行的pull。

问题1：

这个长的镜像ID是在本地build时随机生成的，和内容没有关系，所以再次build生成的ID肯定不同。使用docker pull和push判断image layer是根据这个ID判断的。这就存在问题，恶意用户可以伪造ID直接 push 上去，这样以后再有别人同样的ID就上传不上去了。

问题2：

Docker images的tag是可变的，没有办法通过tag来确定唯一的版本，这一点在latest上尤为明显。因为latest是可以自己定义的，如果在 Dockerfile里from latest很可能你过一阵build出来的镜像和之前不一样了，其他tag也存在同样的问题。避免这个问题的通用做法是用代码 commit ID做tag，每次都是按照commit ID作部署，尽量远离latest。

问题3：Python写的。

v2目录，镜像仓库下存储结构：

```

root@58fb11ad52f3:/var/lib/registry/docker/registry/v2# ls -R
.:
blobs repositories

./blobs:
sha256

./blobs/sha256:
37 8e a3 cf

./blobs/sha256/37:
37dfee6fd180df42676e682afb2bc8101df1af66f5e8f9830346ae43b4813062

./blobs/sha256/37/37dfee6fd180df42676e682afb2bc8101df1af66f5e8f9830346ae43b4813062:
data

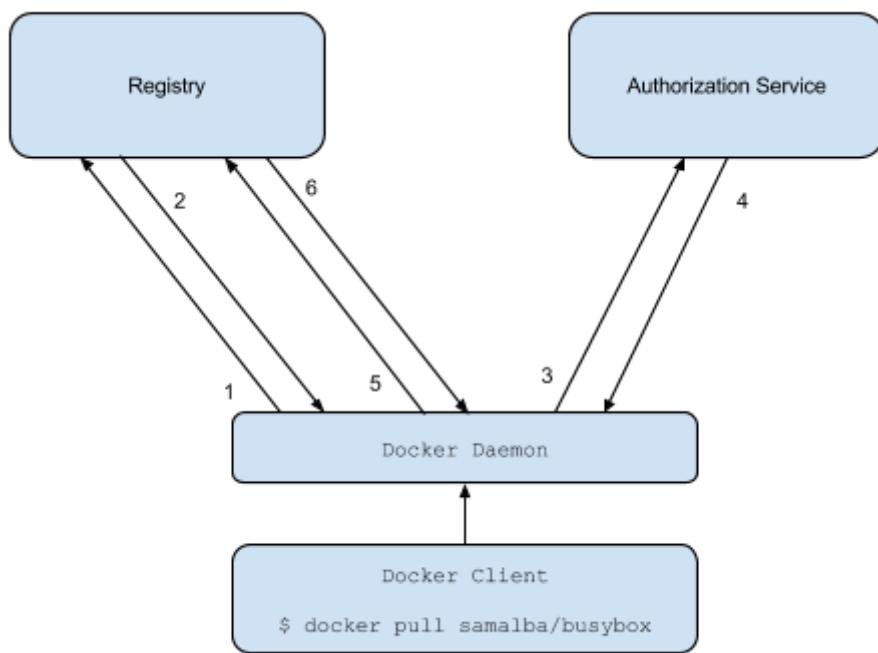
```

最上层是blobs和repositories，blobs和images一样。最叶子节点变为了data，他的目录是一串长ID。这一长串ID（也叫digest）是把data的内容经过sha256做hash得出的结果。这样就很好的解决了之前V1所存在的随机ID的大问题。这样就可以变原来的链表顺序查找为数组的随机读取，这也是V2 pull可以并行的一个基础。

V2中会有一个新的文件，叫manifest，它会记录改镜像所有layer的信息。其他的优点是多了新的Auth方式，通知机制。

V2的缺点：V1提供完整的API，V2提供API的缺失，delete、search这些基本功能都没有。

目前docker registry v2 认证分为以下6个步骤：



1. docker client 尝试到registry中进行push/pull操作；
2. 如果registry的访问需要认证， registry就会返回一个含有如何完成认证的401 Unauthorized HTTP响应；
3. 客户端向Auth服务器请求一个Bearer token；
4. 认证服务器返回给客户端一个加密的Bearer token，用来代表客户端被授权的访问权限；
5. 客户端再次尝试用头部嵌有Bearer token的请求向原来的registry发起请求；
6. registry验证客户端请求中的Bearer token以及包含的授权空间权限。如果正确，便建立与客户端的pull/push会话。

Docker存储

Docker镜像是由一系列的只读层组合而来的，当启动一个容器时，Docker加载镜像的所有只读层，并在最上层加上一个读写层。这种设计使得镜像构建、存储和分发效率很高，但存在问题：多个容器之间的数据无法共享；当删除容器时，容器产生的数据将丢失。为解决这些问题，Docker引入了volume机制。可以为容器添加volume，volume是存在于一个或多个容器中的特定文件或文件夹。volume能在不同容器间共享和重用；对volume中数据的操作会马上生效；volume的生命周期独立于容器的生存周期，删除容器，volume仍然会存在。Docker提供了volumedriver接口，通过实现该接口，可以为Docker容器提供不同的volume存储支持。

为了支持镜像分层与写时复制机制，Docker提供了存储驱动的接口。存储驱动根据操作系统底层的支持提供了针对某种文件系统的初始化操作以及对镜像层的增、删、查和差异比较等操作。目前存储系统的接口有aufs、devicemapper、overlay、zfs、btrfs等。只有vfs不支持写时复制。

aufs存储驱动、devicemapper存储驱动、overlay存储驱动比较

- AUFS是一种Union FS，是文件级的存储驱动。支持将不同目录挂载到同一个虚拟文件系统下的文件系统。这种文件系统可以一层一层地叠加修改文件。无论底下有多少层都是只读的，只有最上层的文件系统是可写的。当需要修改一个文件时，AUFS创建该文件的一个副本，使用写时复制技术将文件从只读层复制到可写层进行修改，结果也保存在可写层。在Docker中，底下的只读层就是image，可写层就是Container。
- Overlay也是一种Union FS，和AUFS的多层不同的是Overlay只有两层：一个lower文件系统和一个upper文件系统，分别代表Docker的镜像层和容器层。当需要修改一个文件时，使用写时复制技术将文件从只读的lower复制到可写的upper进行修改，结果也保存在upper层。在Docker中，底下的只读层就是image，可写层就是Container。
- Device mapper是块级存储，所有的操作都是直接对块进行操作，而不是文件。Device mapper驱动会先在块设备上创建一个资源池，然后在资源池上创建一个带有文件系统的基本设备，所有镜像都是这个基本设备的快照，而容器则是镜像的快照。所以在容器里看到文件系统是资源池上基本设备的文件系统的快照，并没有为容器分配空间。当要写入一个新文件时，在容器的镜像内为其分配新的块并写入数据。当要修改已有文件时，再使用写时复制技术为容器快照分配块空间，将要修改的数据复制到在容器快照中新的块里再进行修改。

Device mapper 驱动默认会创建一个100G的文件包含镜像和容器。每一个容器被限制在10G大小的卷内，可以自己配置调整。

AUFS和Overlay对比：

AUFS和Overlay都是联合文件系统，但AUFS有多层，而Overlay只有两层，所以在做写时复制操作时，如果文件比较大且存在比较低的层，则AUFS可能会慢一些。AUFS在读的方面性能相比Overlay要差一些，但在写的方面性能比Overlay要好。

Overlay和Device mapper对比：

Overlay是文件级存储，Device mapper是块级存储，当文件特别大而修改的内容很小，Overlay不管修改的内容大小都会复制整个文件，对大文件进行修改显示要比小文件要消耗更多的时间，而块级无论是大文件还是小文件都只复制需要修改的块到文件(默认就一个100G的文件)里，并不是整个文件，在这种场景下，显然device mapper要快一些。是对块操作，不支持共享存储，表示当有多个容器读同一个文件时，需要生成多个副本，所以这种存储驱动不适合在高密度容器的PaaS平台上使用。Device mapper在512M以上文件的读写性能都非常的差，但在512M以下的文件读写性能都比较好。

Docker数据卷

为容器添加volume，类似于Linux的mount操作，用户将一个文件夹作为volume挂载到容器上。多个容器可以共享同一个volume，为不同容器之间的数据共享提供便利。

docker1.9后两种办法创建volume。

1.不用docker提供的命令，直接使用

```
1. #将宿主机上/usr/local目录挂载到容器内/local目录
2. docker run -itd -v /user/local:/local /bin/bash
```

2.使用docker提供的volume命令。docker volume会在宿主机/var/lib/docker/volume/[volumeid]/_data创建目录，以后对volume的内容都存储在该目录下。

```
1. [root@localhost cpu]# docker volume create --name hello-volume
2. hello-volume
3. [root@localhost cpu]#
4. [root@localhost cpu]# docker volume inspect hello-volume
5. [
6.     {
7.         "Name": "hello-volume",
8.         "Driver": "local",
9.         "Mountpoint": "/var/lib/docker/volumes/hello-volume/_data",
10.        "Labels": {},
11.        "Scope": "local"
12.    }
13. ]
14. [root@localhost cpu]#
15. [root@localhost cpu]# docker run -it --name volumetest -v hello-volume:/volume centos:7
```

可以使用--volumes-from标签使得容器与已有的容器共享volume。

```
1. docker run -it --name volumetest2 --volumes-from volumetest centos:7 /bin/bash
```

删除volume

```
1. docker volume rm hello-volume
```

网络

Docker在1.9版本中引入了一整套的docker network子命令和跨主机网络支持。这允许用户可以根据他们应用的拓扑结构创建虚拟网络并将容器接入其所对应的网络。Docker1.7中，网络部分代码就已经被抽离并单独成为了Docker的网络库，即libnetwork。在此之后，容器的网络模式也被抽象变成了统一接口的驱动。

为了标准化网络驱动的开发步骤和支持多种网络驱动，Docker公司在libnetwork中使用了CNM（Container Network Model）。CNM定义了构建容器虚拟化网络的模型，同时还提供了可以用于开发多种网络驱动的标准化接口和组件。

libnetwork和Docker daemon及各个网络驱动的关系如下图表示：

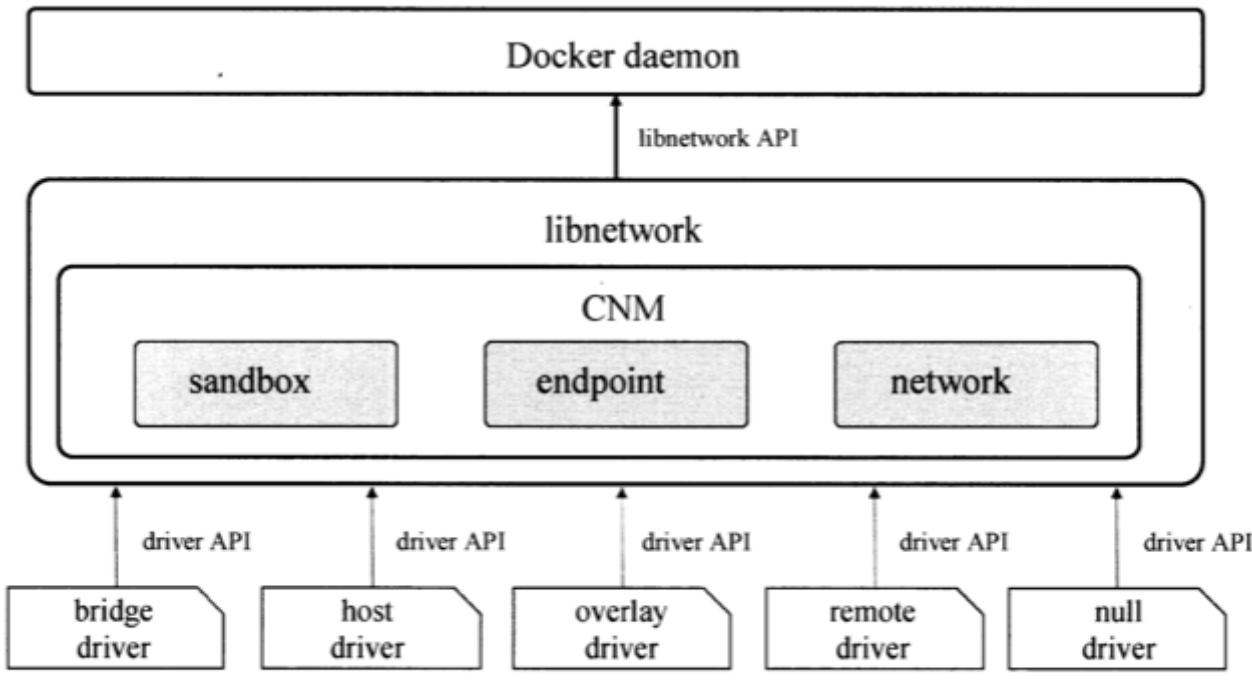


图3-16 Docker网络虚拟化架构

Docker daemon通过调用libnetwork对外提供的API完成网络的创建和管理等功能。libnetwork中则使用了CNM来完成网络功能的提供。而CNM中主要有沙盒（sandbox）、端点（endpoint）和网络（network）这3种组件。libnetwork中内置的5种驱动则为libnetwork提供了不同类型的网络服务。下面分别对CNM中的3个核心组件和libnetwork中的5种内置驱动进行介绍。

CNM中的3个核心组件如下：

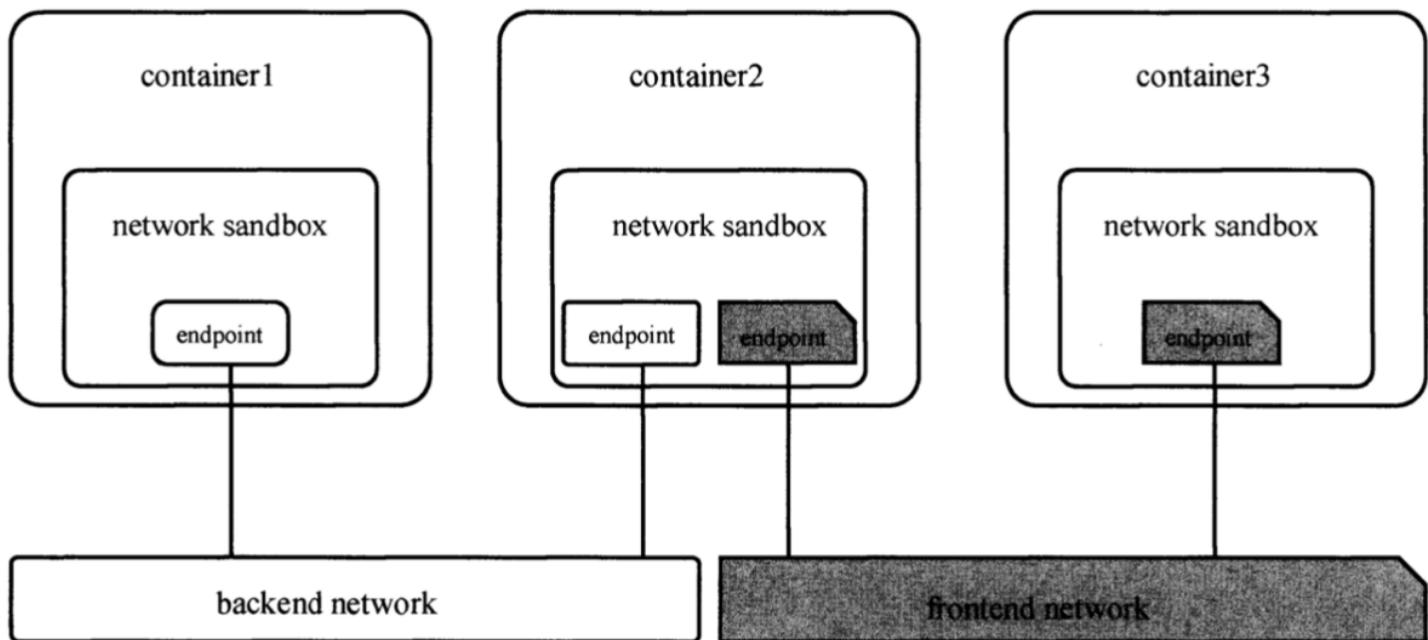
- 沙盒：一个沙盒包含了一个容器网络栈的信息。沙盒可以对容器的接口、路由和DNS设置等进行管理。沙盒的实现可以使用linux network namespace。一个沙盒可以有多个端点和多个网络。
- 端点：一个端点可以加入一个沙盒和一个网络。端点的实现可以是veth pair等相似的设备。一个端点只能属于一个网路并且只属于一个沙盒。
- 网络：一个网络是一组可以直接互相联通的端点。网络的实现可以是Linux bridge、VLAN等。一个网络可以包含多个端点。

libnetwork中的5种内置驱动如下：

- bridge驱动 此驱动为Docker的默认设置，使用这个驱动的时候，libnetwork将创建出来的Docker容器连接到Docker网桥上，容器与外界通信使用NAT模式。
- host驱动 使用这种驱动的时候，libnetwork将不为Docker容器创建网络协议栈，即不会创建独立的network namespace。Docker容器中的进程处于宿主机的网络环境中，相当于Docker容器和宿主机公用同一个network namespace，使用宿主机的网卡、IP和端口等信息。但是host驱动也降低了容器与容器之间、容器与宿主机之间网络层面的隔离性，引起网络资源的竞争与冲突，因此可以认为host驱动适用于对于容器集群规模不大的场景。
- overlay驱动 此驱动采用IETF标准的VXLAN方式，叠加网络。比如flannle的CNM实现。
- remote驱动 这个驱动实际上并未做真正的网络服务实现，而是调用了用户自行实现的网络驱动插件，使libnetwork实现了驱动的可插件化，更好地满足了用户的多种需求。用户只要根据libnetwork提供的协议标准，实现其所要求的各个接口并向Docker daemon进行注册。
- null驱动 Docker容器拥有自己的network namespace，但是并不为Docker容器进行任何网络配置。也就是说，这个Docker容器除了network namespace自带的loopback网卡外，没有其他任何网卡、IP、路由等信息，需要用户为Docker容器添加网卡、配置IP等。这种模式如果不进行特定的配置是无法正常使用的，但是优点也非常明显，它给了用户最大的自由度来自定义容器的网络环境。leengien就用了。直接创建一个veth设备对，一端直接放到容器的net ns中。

在引入libnetwork后，现在用户可以利用Docker的网络命令创建更多与默认网络相似的网络，每一个都是特定类型网络插件的实体。

演示libnetwork。



他有两个网络，其中backend network为后端网络，frontend network为前端网络，两个网络不通。其中container1和container3各拥有一个端点veth，并且分别加入到后端网络和前端网络中。而container2则有两个端点，它分别加入到后端网络和前端网络中。

演示：

```
1. #这三个网络是docker daemon默认创建的，分别使用了不同的驱动。
2. [root@localhost docker]# docker network ls
3. NETWORK ID      NAME      DRIVER      SCOPE
4. 2d05b26c275b    bridge    bridge      local
5. c2ff85e08519    host      host       local
6. f868849d8b13    none     null       local
7. [root@localhost docker]# docker network create backend
8. fd36fe1949702106cd2dae3f8277f6ce4f5c03c517a12eab201a731c546cd495
9. [root@localhost docker]# docker network create frontend
10. 437322d8dbaf8eef8f7c2e6a34b7a667a9be923b0028cbfef8a02f78c7dce22
11. [root@localhost docker]# docker network ls
12. NETWORK ID      NAME      DRIVER      SCOPE
13. fd36fe194970    backend   bridge      local
14. 2d05b26c275b    bridge    bridge      local
15. 437322d8dbaf    frontend  bridge      local
16. c2ff85e08519    host      host       local
17. f868849d8b13    none     null       local
18. [root@localhost docker]#
19. #将container1和container2假如到backend网络，将container3假如到frontend网络
20. [root@localhost docker]# docker run -it --name container1 --net backend busybox
21. / # [root@localhost docker]#
22. [root@localhost docker]# docker run -it --name container2 --net backend busybox
23. / # [root@localhost docker]#
24. [root@localhost docker]# docker run -it --name container3 --net frontend busybox
25. / # [root@localhost docker]#
26. [root@localhost docker]#
27. #查看container1、2、3的IP，互ping，发现只有在同一个网络的才能ping通。
28. [root@localhost ~]# docker exec container1 ip addr
```

```
29. 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
30.     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
31.     inet 127.0.0.1/8 scope host lo
32.         valid_lft forever preferred_lft forever
33.     inet6 ::1/128 scope host
34.         valid_lft forever preferred_lft forever
35. 10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
36.     link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
37.     inet 172.18.0.2/16 scope global eth0
38.         valid_lft forever preferred_lft forever
39.     inet6 fe80::42:acff:fe12:2/64 scope link
40.         valid_lft forever preferred_lft forever
41. [root@localhost ~]# docker exec container2 ip addr
42. 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
43.     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
44.     inet 127.0.0.1/8 scope host lo
45.         valid_lft forever preferred_lft forever
46.     inet6 ::1/128 scope host
47.         valid_lft forever preferred_lft forever
48. 12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
49.     link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
50.     inet 172.18.0.3/16 scope global eth0
51.         valid_lft forever preferred_lft forever
52.     inet6 fe80::42:acff:fe12:3/64 scope link
53.         valid_lft forever preferred_lft forever
54. [root@localhost ~]# docker exec container2 ping 172.18.0.2
55. PING 172.18.0.2 (172.18.0.2): 56 data bytes
56. 64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.058 ms
57. 64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.106 ms
58. ^C
59. [root@localhost ~]# docker exec container3 ip addr
60. 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
61.     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
62.     inet 127.0.0.1/8 scope host lo
63.         valid_lft forever preferred_lft forever
64.     inet6 ::1/128 scope host
65.         valid_lft forever preferred_lft forever
66. 14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
67.     link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff
68.     inet 172.19.0.2/16 scope global eth0
69.         valid_lft forever preferred_lft forever
70.     inet6 fe80::42:acff:fe13:2/64 scope link
71.         valid_lft forever preferred_lft forever
72. [root@localhost ~]# docker exec container2 ping 172.19.0.2
73. ^C
74. [root@localhost ~]#
75. #在container2中使用命令ifconfig来查看此容器中的网卡及其配置情况。可以看到，此容器中只有一块以太网卡，其
76. [root@localhost ~]# docker exec container2 ifconfig
77. eth0      Link encap:Ethernet HWaddr 02:42:AC:12:00:03
78.           inet addr:172.18.0.3 Bcast:0.0.0.0 Mask:255.255.0.0
79.             inet6 addr: fe80::42:acff:fe12:3/64 Scope:Link
80.               UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
81.               RX packets:286 errors:0 dropped:0 overruns:0 frame:0
82.               TX packets:516 errors:0 dropped:0 overruns:0 carrier:0
83.               collisions:0 txqueuelen:0
84.               RX bytes:26380 (25.7 KiB) TX bytes:48976 (47.8 KiB)
85. lo        Link encap:Local Loopback
86.           inet addr:127.0.0.1 Mask:255.0.0.0
87.             inet6 addr: ::1/128 Scope:Host
88.               UP LOOPBACK RUNNING MTU:65536 Metric:1
89.               RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```

90.          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
91.          collisions:0 txqueuelen:1
92.             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
93.
94. #将container2加入到frontend网络中
95. [root@localhost ~]# docker network connect frontend container2
96. #docker network connect命令会在所连接的容器中创建新的网卡，以完成其与所指定网络的连接。
97. [root@localhost ~]# docker exec container2 ifconfig
98. eth0      Link encap:Ethernet HWaddr 02:42:AC:12:00:03
99.         inet addr:172.18.0.3 Bcast:0.0.0.0 Mask:255.255.0.0
100.        inet6 addr: fe80::42:acff:fe12:3/64 Scope:Link
101.           UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
102.           RX packets:317 errors:0 dropped:0 overruns:0 frame:0
103.           TX packets:572 errors:0 dropped:0 overruns:0 carrier:0
104.           collisions:0 txqueuelen:0
105.           RX bytes:29250 (28.5 KiB)  TX bytes:54296 (53.0 KiB)
106. eth1      Link encap:Ethernet HWaddr 02:42:AC:13:00:03
107.         inet addr:172.19.0.3 Bcast:0.0.0.0 Mask:255.255.0.0
108.         inet6 addr: fe80::42:acff:fe13:3/64 Scope:Link
109.           UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
110.           RX packets:9 errors:0 dropped:0 overruns:0 frame:0
111.           TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
112.           collisions:0 txqueuelen:0
113.           RX bytes:746 (746.0 B)  TX bytes:746 (746.0 B)
114. lo        Link encap:Local Loopback
115.         inet addr:127.0.0.1 Mask:255.0.0.0
116.         inet6 addr: ::1/128 Scope:Host
117.           UP LOOPBACK RUNNING MTU:65536 Metric:1
118.           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
119.           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
120.           collisions:0 txqueuelen:1
121.           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
122. [root@localhost ~]#
123. #测试container2与container3的连通性后，可以发现两者已经互通
124. root@localhost ~]# docker exec container2 ping 172.19.0.2
125. PING 172.19.0.2 (172.19.0.2): 56 data bytes
126. 64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.063 ms
127. 64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.069 ms
128. ^C
129. [root@localhost ~]#

```

关于Linux bridge、Veth设备对和network namespace的补充：

- Linux Bridge，即Linux网桥设备，是Linux提供的一种虚拟网络设备之一。其工作方式非常类似于物理的网络交换机设备。Linux Bridge可以工作在二层，也可以工作在三层，默认工作在二层。工作在二层时，可以在同一网络的不同主机间转发以太网报文；一旦你给一个Linux Bridge分配了IP地址，也就开启了该Bridge的三层工作模式。在Linux下，你可以用 iproute2 工具包或brctl命令对Linux bridge进行管理。
- VETH(Virtual Ethernet)虚拟网络设备对是Linux提供的另外一种特殊的网络设备，它总是成对出现，要创建就创建一个pair。一个Pair中的veth就像一个网络线缆的两个端点，数据从一个端点进入，必然从另外一个端点流出。每个veth都可以被赋予IP地址，并参与三层网络路由过程。两个不同命名空间默认是给的，要实现通信，只能使用一对Veth设备对，各自放在两个network namespace。
- Network namespace，网络名字空间，允许你在Linux创建相互隔离的网络视图，每个网络名字空间都有独立的网络配置，比如：网络设备、路由表等。新建的网络名字空间与主机默认网络名字空间之间是隔离的。我们平时默认操作的是主机的默认网络名字空间。

可以使用网络命名空间和Veth模拟Docker的网络，两个容器互通。

Docker0软网桥的“双重身份”，docker0将在这两种身份间来回切换。

1、从容器视角，网桥（交换机）身份

docker0对于通过veth pair“插在”网桥上的container1和container2来说，首先就是一个二层的交换机的角色，在二层转发数据包；同时由于docker0自身也具有mac地址（这个与纯二层交换机不同），并且绑定了ip（这里是172.17.0.1），因此在 container中还作为container default路由的默认Gateway而存在。

2、从宿主机视角，网卡身份

所有docker0从veth（只是个二层的存在，没有绑定ipv4地址）接收到的数据包都会被宿主机看成从docker0这块网卡（第二个身份，绑定172.17.0.1）接收进来的数据包，尤其是在进入三层时，宿主机上的iptables就会对docker0进来的数据包按照rules进行相应处理（通过一些内核网络设置也可以忽略docker0 brigde数据的处理）。

Docker针对端口映射前后有两种方案：

- 一种是1.7版本之前docker-proxy+iptables DNAT 的方式；
- 另一种则是1.7版本(及之后)提供的完全由iptables DNAT实现的端口映射。

不过在目前docker 1.9.1中，前一种方式依旧是默认方式。但是从Docker 1.7版本起，Docker提供了一个配置项：–userland-proxy，以让Docker用户决定是否启用docker-proxy，默認為true，即启用docker-proxy。

我们修改了一下/etc/default/docker配置，为DOCKER_OPTS增加一个option: –userland-proxy=false。

```
1. DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --userland-proxy=false"
```

由于每个做端口映射的Container都要启动至少一个docker proxy与之配合，一旦运行的container增多，那么多个docker proxy进程对资源的消耗将是大大的。因此docker engine在docker 1.7之后提供了完全由iptables DNAT实现的端口映射，无需再启动docker proxy进程。我们只需修改一下docker engine的启动配置即可。

Docker安装完成后，将默认在宿主机系统上增加一些iptables规则，以用于Docker容器和容器之间以及和外界的通信，可以使用iptables-save命令查看。其中nat表上的POSTROUTING链有这么一条规则：

```
1. -A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
```

同宿主机容器互联

容器间通信由Docker daemon的启动参数–icc控制。–icc为true，保证默认可以互联。但为了保证容器以及主机的安全，–icc通常设置为false。这种情况下该如何解决容器间的通信呢？最长用的方式是端口映射，这种方式不够安全，而且需要经过NAT，效率也不高。这时候就需要Docker连接（link）。Docker的连接系统可以在两个容器之间建立一个安全的通道，使得接受容器（如Web容器）可以通过通道得到容器（如数据库服务）指定的相关信息。

在Docker1.9版本后，网络操作独立出命令组（docker network），link方式也与原来不同了。Docker为了向上兼容，若容器使用默认的bridge模式网络，则会默认使用传统的link系统；而使用用户自定义的网络，则会使用新的link系统。

- 传统link方式：

```
1. docker run -d --name db training/postgres
2. docker run -d -P --name web --link db:webdb training/webapp python app.py
```

--link格式是--link 容器名:别名。

那么--link到底做了什么呢？

1. 设置接受容器的环境变量。

我们发现在接收容器中有一组关于容器db的环境变量。有几个link，就多几组。

```
1. WEBDB_PORT_8080_TCP_ADDR=172.17.0.82
2. WEBDB_PORT_8080_TCP_PORT=8080
3. WEBDB_PORT_8080_TCP_PROTO=tcp
4. WEBDB_PORT_8080_TCP=tcp://172.17.0.82:8080
5. WEBDB_PORT=tcp://172.17.0.82:8080
```

2. 更新接收容器的/etc/hosts文件

Docker容器的IP地址是不固定的，当容器重启后，IP地址可能就和之前不同了，但是接收容器中的这一组环境变量还是不变。因此，Docker的link操作除了在将link信息保存在接收容器中之外，还在/etc/hosts中添加了一项——源容器的IP和别名（--link参数中指定的别名），以用来解决源容器的IP地址，当容器重启后会自动更新接收容器的/etc/hosts文件。

3. 紧靠上面两条还不能互相通信。将Docker daemon的-icc参数设置为false时，容器间的同心就被禁止了。这时Docker daemon为了保证两个容器的通信，添加特定的iptables规则。

```
1. -A DOCKER -s 172.17.0.2/32 -d 172.17.0.1/32 -i docker0 -o docker0 -p tcp -m tcp --dport 5
2. -A DOCKER -s 172.17.0.1/32 -d 172.17.0.2/32 -i docker0 -o docker0 -p tcp -m tcp --dport 5
```

这两条规则确保了web容器和db容器在db容器的tcp/5432端口上通信的流量不会被丢弃掉，从而保证了接受容器可以顺利地从源容器中获取想要的数据。

- 新的link方式：

Docker 1.9后为用户自定义网络提供了DNS自动名字解析、同一个网络中容器间的隔离、可以动态加入或者退出多个网络、支持--link为源容器设定别名等服务。

新的网络模型中，link不要求源容器已经创建或者启动，只是在当前网络给源容器起了个别名，并且这个别名只对接收容器有效。

```
1. [root@localhost docker]# docker network create isolated_nw
2. fe1a11931fbfe06c7676d27f51f8795c6e1cbcd2ae07f8192da664691e307ac
3. [root@localhost docker]# docker run --net=isolated_nw -it --name=container1 --link container2 busybox
4. / # [root@localhost docker]#
5. [root@localhost docker]# docker run --net=isolated_nw -itd --name=container2 busybox
6. 0b5cc8ee03c6c20418c5c42a0bead204cc4fbfd8ce05396e79bf88ffffb3c9759
7. [root@localhost docker]# docker exec container1 ping c2
8. PING c2 (172.20.0.3): 56 data bytes
9. 64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.061 ms
10. 64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.065 ms
11. 64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.103 ms
12. ^C
13. [root@localhost docker]#
```

另外在container1下查看/etc/hosts文件后，发现里面并没有container2的相关信息，这表新的link系统的实现与原来的配置hosts文件的方式并不相同。实际上，Docker是通过DNS解析的方式提供名字和别名的解析。

跨主机集群容器网络

使用4种网络驱动的具体情况如下：

- 使用host确定可以让容器与宿主机公用一个网络栈，这么做看似解决了网络问题，可实际上并未使用network namespace的隔离，缺乏安全性。
- 使用Docker默认的bridge驱动，容器没有对外IP，只能通过NAT来实现对外通信。这种方式不能解决跨主机容器间直接通信的问题。
- 使用overlay驱动，可以用于支持跨主机的网络通信，但必须要配合Swarm进行配置和使用才能实现跨主机的网络通信。在Docker 1.9版本之前，社区中就已经有许多第三方的工具或方法尝试解决这个问题，例如

Macvlan、Pipework、Flannel、Weave等。虽然这些方案在实现细节上存在很多差异，但其思路无非分为两种：二层VLAN网络和Overlay网络。

- 使用null驱动，实际上不进行任何网络设置。

Docker 1.9以后再讨论容器网络方案（因为Docker增加了Overlay驱动插件，第三方可以加入），不仅要看实现方式，而且还要看网络模型的“站队”，比如说你到底是要用Docker原生的“CNM”，还是CoreOS，谷歌主推的“CNI”。

Docker Libnetwork Container Network Model (CNM) 阵营

Docker Swarm overlay

Macvlan & IP network drivers

Calico

Contiv (from Cisco)

Docker Libnetwork的优势就是原生，而且和Docker容器生命周期结合紧密；缺点也可以理解为是原生，被Docker“绑架”。

Container Network Interface (CNI) 阵营

Kubernetes

Weave

Macvlan

Flannel

Calico

Contiv

Mesos CNI

CNI的优势是兼容其他容器技术 (e.g. rkt) 及上层编排系统 (Kubernetes & Mesos)，而且社区活跃势头迅猛，Kubernetes加上CoreOS主推；缺点是非Docker原生。

跨主机网络通信方案：

1. 利用虚拟网桥将Docker容器桥接到本地网络

本地网络为10.10.103.0/24，网关为10.10.103.254，要将容器test1的IP地址配置为10.10.103.91/24（网卡为eth0）：

```
1. [root@localhost ~]# ip addr
2. 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
   link/ether 52:54:00:88:15:b6 brd ff:ff:ff:ff:ff:ff
   inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic eth0
      valid_lft 85824sec preferred_lft 85824sec
   inet6 fe80::5054:ff:fe88:15b6/64 scope link
      valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
   link/ether 02:42:13:40:9f:77 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.1/16 scope global docker0
      valid_lft forever preferred_lft forever
5: [root@localhost ~]#
6: [root@localhost ~]# 1. 启动容器, 设置net=none
```

```
20. [root@localhost ~]# docker run -itd --name test1 --net=none centos /bin/bash
21. 3474ee9a3fde4f473c38af597a7f355be6b3c2148c5a4766ca6ce55c89954122
22. [root@localhost ~]#
23. [root@localhost ~]#2. 创建供容器使用的网桥br0
24. [root@localhost ~]# brctl addbr br0
25. [root@localhost ~]# ip link set br0 up
26. [root@localhost ~]#3. 将主机eth0桥接到br0上，并把eth0的IP配置在br0上。由于是远程操作，会导致网络断开
27. [root@localhost ~]# ip addr add 10.0.2.15/24 dev br0; \
28.     ip addr del 10.0.2.16/24 dev eth0; \
29.     brctl addif br0 eth0; \
30.     ip route del default; \
31.     ip route add default via 10.0.2.255 dev br0
RTNETLINK answers: Cannot assign requested address
RTNETLINK answers: Network is unreachable
32. [root@localhost ~]#
33. [root@localhost ~]#
34. [root@localhost ~]#4. 找到test1的PID，保存到pid中
35. [root@localhost ~]# pid=$(docker inspect --format '{{.State.Pid}}' test1)
36. [root@localhost ~]# 5. 将容器的network namespace添加到/var/run/netns目录下
37. [root@localhost ~]# mkdir -p /var/run/netns
38. [root@localhost ~]# ln -s /proc/$pid/ns/net /var/run/netns/$pid
39. [root@localhost ~]#
40. [root@localhost ~]#6. 创建用于连接网桥和Docker容器的网卡设备，将veth-a连接到br0网桥上，veth-b放入容器
41. [root@localhost ~]# ip link add veth-a type veth peer name veth-b
42. [root@localhost ~]# brctl addif br0 veth-a
43. [root@localhost ~]# ip link set veth-a up
44. [root@localhost ~]# 7. 将veth-b放到test1的network namespace中，重命名为eth0，并为其配置IP和默认网关
45. [root@localhost ~]# ip link set veth-b netns $pid
46. [root@localhost ~]# ip netns exec $pid ip link set dev veth-b name eth0
47. [root@localhost ~]# ip netns exec $pid ip link set eth0 up
48. [root@localhost ~]# ip netns exec $pid ip addr add 10.0.2.16/24 dev eth0
49. [root@localhost ~]# ip netns exec $pid ip route add default via 10.0.2.255
50. RTNETLINK answers: Network is unreachable
51. [root@localhost ~]#
52. [root@localhost ~]#
53. [root@localhost ~]#在容器内能拼通容器外宿主机
54. [root@7b7799616fa8 /]# ping 10.0.2.15
55. PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
56. 64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.199 ms
57. 64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.063 ms
58. 64 bytes from 10.0.2.15: icmp_seq=3 ttl=64 time=0.059 ms
59. ^C
60. --- 10.0.2.15 ping statistics ---
61. 3 packets transmitted, 3 received, 0% packet loss, time 2001ms
62. rtt min/avg/max/mdev = 0.059/0.107/0.199/0.065 ms
63. [root@7b7799616fa8 /]# [root@localhost ~]#
64. [root@localhost ~]# ping 10.0.2.16
65. PING 10.0.2.16 (10.0.2.16) 56(84) bytes of data.
66. 64 bytes from 10.0.2.16: icmp_seq=1 ttl=64 time=0.039 ms
67. 64 bytes from 10.0.2.16: icmp_seq=2 ttl=64 time=0.069 ms
68. 64 bytes from 10.0.2.16: icmp_seq=3 ttl=64 time=0.056 ms
69. ^C
70. --- 10.0.2.16 ping statistics ---
71. 3 packets transmitted, 3 received, 0% packet loss, time 1999ms
72. rtt min/avg/max/mdev = 0.039/0.054/0.069/0.014 ms
73. [root@localhost ~]#
74. 但在其他主机上拼不通这个IP
75. → ~ ping 10.0.2.16
76. PING 10.0.2.16 (10.0.2.16): 56 data bytes
77. Request timeout for icmp_seq 0
78. Request timeout for icmp_seq 1
```

```
81. Request timeout for icmp_seq 2
82. Request timeout for icmp_seq 3
83. ^C
84. --- 10.0.2.16 ping statistics ---
85. 8 packets transmitted, 0 packets received, 100.0% packet loss
86. → ~
87. → ~
```

2. 使用pipework工具配置各种类型网络

2.1 将Docker容器与宿主机配置在本地的网络环境中

```
1. #下载pipework
2. $ git clone https://github.com/jpetazzo/pipework
3. $ 将pipework脚本放入PATH环境变量所指定的目录下, 如/usr/local/bin
4. $ cp ~/pipework/pipework /usr/local/bin/
5. $
6. # 完成test1的配置
7. $ pipework br0 test1 10.10.103.95/24@10.10.103.254
```

这一行配置命令执行的操作如下：

- 查看主机是否存在br0网桥，不存在就创建；
- 向容器test1中加入一块名为eth1的网卡，并设置IP地址为10.10.103.95/24；
- 若容器test1中已经有默认路由，则删掉，把10.10.103.254设为默认路由的网关；
- 将容器test1连接到之前创建的网桥br0上。

这个过程和上面手动过程一样。

2.2 支持使用macvlan设备将容器连接到本地网络

也可以使用macvlan将Docker容器连接到本地网络，macvlan设备是从网卡上虚拟出一块新网卡，它和主网卡分别有不同的MAC地址，可以配置独立的IP地址。目前Docker网络本身不提供macvlan支持，但可以借助pipework来完成macvlan配置。

整个过程只需要执行一条命令：

```
1. $ pipework eth0 test1 10.10.103.95/24@10.10.103.254
```

这里，pipework的参数1 eth0是主机上的一块以太网卡，而非网桥。pipework采用macvlan设备作为test1容器的网卡，不会再创建veth pair设备来连接容器和网桥，操作过程如下：

- 从主机的eth0上创建一块macvlan设备，将macvlan设备放入到test1容器中并命名为eth1；
- 为容器test1中新添加的网卡配置IP地址为10.10.103.95/24；
- 若容器test1中已经有默认路由，则删掉，把10.10.103.254设为默认路由的网关。

3 通过外部DHCP服务器获取并设置容器的IP

如果Docker要介入的网络环境中存在DHCP服务器，那么Docker容器就可以通过发送DHCP请求获取新网卡的网络配置信息。具体用法是将pipework指令中的IP地址参数替换为dhcp，示例如下：

```
1. #配置直接将宿主机eth0的macvlan的子设备作为容器test1的网卡, 为新网卡设置IP, 设置默认路由的网关
2. pipework eth0 test1 10.10.103.95/24@10.10.103.254
3. 通过主机网络中的DHCP服务器获取IP地址的命令
4. pipework eth0 test1 dhcp
```

上面的3个方法都可以将Docker容器连接到本地网络环境中，之间可以直接通信。但这么做可能会出现下列问题：

- Docker容器占用主机网络的IP地址；
- 大量Docker容器可能引起广播风暴，导致主机所在网络性能的下降；
- Docker容器连在主机网络中可能引起安全问题。

如果必须将Docker容器连接在主机网络中，最好还是将其分离开，不再一个大二层网络中。为了隔离Docker容器间网络和主机网络，需要额外使用一块网卡桥接Docker容器。思路：

在所有主机上用虚拟网桥（如docker0网桥）将本机的Docker容器连接起来，然后将一块网卡（eth1）加入到虚拟网桥中，eth1不需要配置IP，使所有主机上的虚拟网桥级联在一起，这样，不同主机上的Docker容器也就如同连在一个大的逻辑交换机上。这就是桥接。

关于Docker容器的IP，由于不同机器上的Docker容器可能获得相同的IP地址，因此需要解决IP冲突，我们为每一台主机上的Docker daemon指定不同的--fixed-cidr参数，将不同主机上的Docker容器的地址限定在不同的网段中。

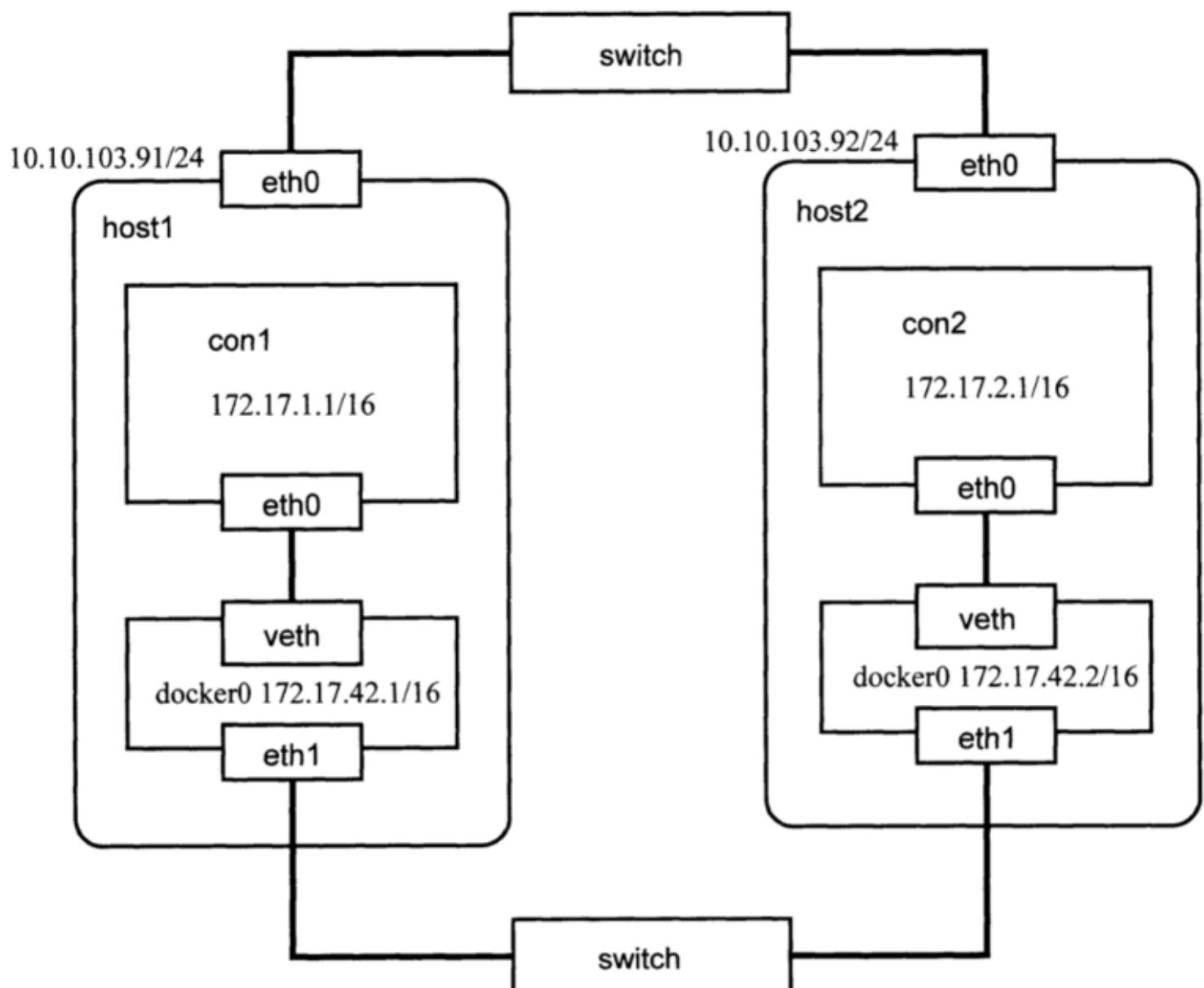


图4-3 桥接网络拓扑图

```
1. #在host1上作如下操作
2. $ echo 'DOCKER_OPTS="--fixed-cidr=172.17.1.1/24"' >> /etc/default/docker
3. $ service docker stop
4. $ service docker start
```

```
5. #将eth1网卡接入到docker0网桥中
6. $ brctl addif docker0 eth1
7.
8. #在host2上作如下操作
9. $ echo 'DOCKER_OPTS="--fixed-cidr=172.17.2.1/24"' >> /etc/default/docker
10. #为避免和host1的docker0的IP冲突, 修改docker0的IP
11. $ ifconfig docker0 172.17.42.2/16
12. $ service docker stop
13. $ service docker start
14. #将eth1网卡接入到docker0网桥中
15. $ brctl addif docker0 eth1
```

容器con1（172.17.1.1）向容器con2（172.17.2.1）发送数据的过程是这样的：首先，通过查看本身路由表发现目的地址和自己处于同一个网段，子网掩码是16(255.255.0.0)，那么就不需要将数据发往网关，可以直接发给con2，con1通过ARP广播获取到con2的MAC地址；然后，构建以太网帧发往con2即可。此过程数据流经的路径如上图中两个容器的eth0网卡所连接的路径，其中docker0网桥充当普通的交换机转发数据帧。

3 直接路由

桥接方式是将所有主机上的Docker容器放在一个二层网络中，它们之间通信是由交换机直接转发，不通过路由器。另一种跨主机通信的方式是通过在主机中添加静态路由实现的。如果有两台主机host1和host2，两主机上的Docker容器是两个独立的二层网络，将con1发往con2的数据流先转发到主机host2上，再由host2再转发到其上的Docker容器中；反之亦然。

桥接网络是二层通信，通过MAC地址转发；直接路由为三层通信，通过IP地址进行路由转发。

由于使用容器的IP进行路由，就需要避免不同主机上的Docker容器使用相同的IP，所以应该为不同的主机分配不同的IP子网。

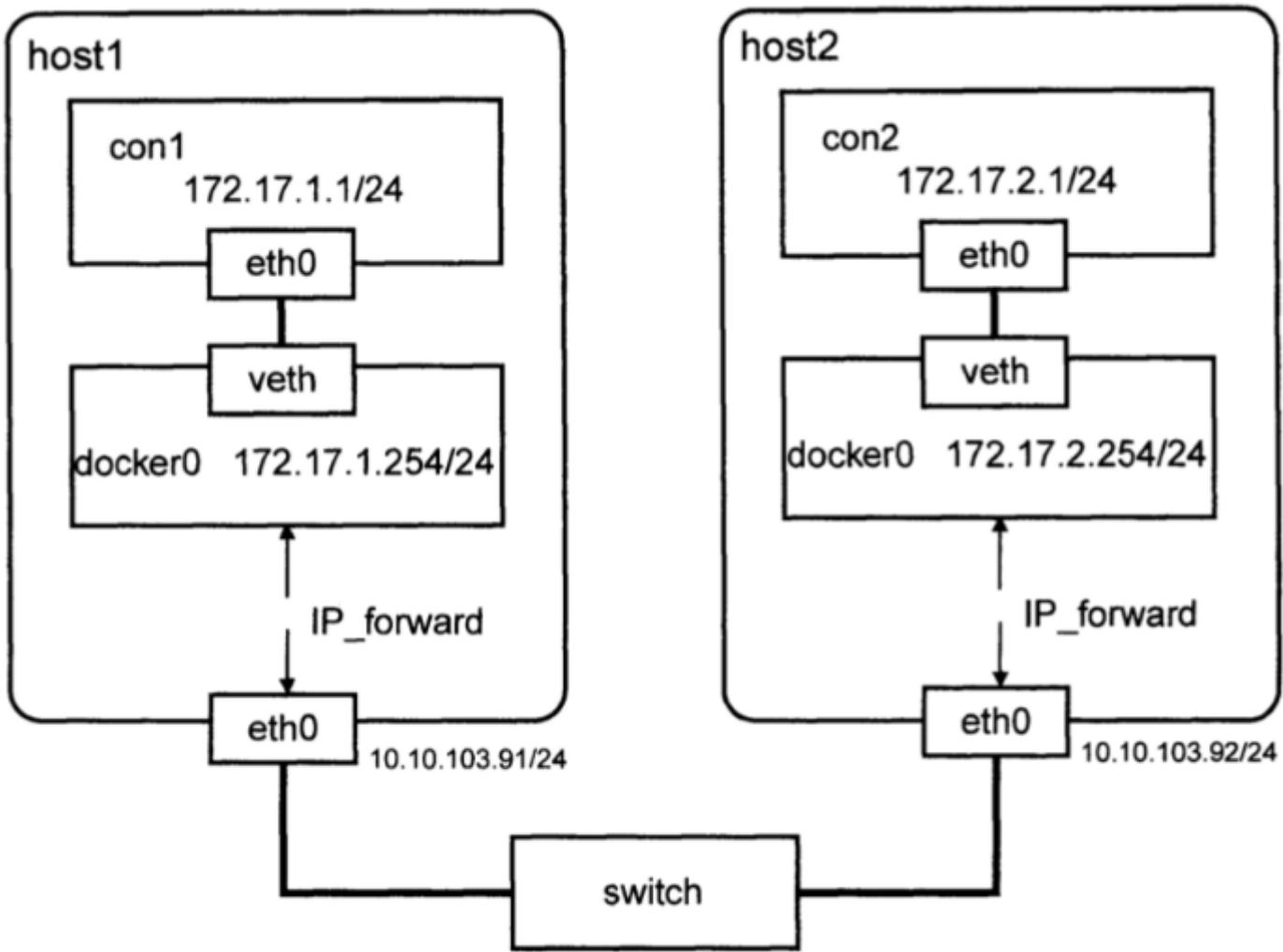


图4-4 路由网络拓扑图

```

1. #在host1上做如下操作
2. #为防止与不同主机的IP冲突, 影响配置路由规则, 配置docker0的IP地址
3. $ ifconfig docker0 172.17.1.254/24
4. $ service docker restart
5. #添加路由, 将目的地址为172.17.2.0/24的包转发到host2
6. $ route add -net 172.17.2.0 netmask 255.255.255.0 gw 10.10.103.92
7. #配置iptables规则
8. $ iptables -t nat -F POSTROUTING
9. $ iptables -t nat -A POSTROUTING -s 172.17.1.0/24 ! -d 172.17.0.0/16 -j MASQUERADE
10. #启动容器con1
11. $ docker run -it --name con1 ubuntu /bin/bash
12. #在con1容器中
13. #nc -l 9000
14.
15.

16. #在host2上做如下操作
17. #为防止与不同主机的IP冲突, 影响配置路由规则, 配置docker0的IP地址
18. $ ifconfig docker0 172.17.2.254/24
19. $ service docker restart
20. #添加路由, 将目的地址为172.17.1.0/24的包转发到host1
21. $ route add -net 172.17.1.0 netmask 255.255.255.0 gw 10.10.103.91
22. #配置iptables规则
23. $ iptables -t nat -F POSTROUTING
24. $ iptables -t nat -A POSTROUTING -s 172.17.2.0/24 ! -d 172.17.0.0/16 -j MASQUERADE
25. #启动容器con2
26. $ docker run -it --name con2 ubuntu /bin/bash
27. #在con2容器中

```

```
28. #nc -w 1 -v 172.17.1.1 9000  
29. Connection to 172.17.1.1 9000 port [tcp/*] succeeded!
```

4.1 使用Quagga软件实现路由规则的动态添加

使用Quagga软件有两种办法：可以在每台服务器上安装Quagga软件并启动，还可以下载Quagga容器（配置网络为--net=host，使用宿主机网络）来运行。

启动完所有主机上Quagga容器后，Quagga会互相学习来完成到其他机器的docker0路由规则的添加。

一段时间后，在Node1上使用route -n命令来查看路由表，可以看到Quagga自动添加了两条到Node2和到Node3上docker0的路由规则。在Node2上查看路由表，可以看到自动添加了两条到Node1和Node3上docker0的路由规则。至此，所有Node上的docker0都可以互通了。

4 OVS划分VLAN（使用隧道方案Open vSwitch代替docker0实现Docker容器的VLAN划分（会用到pipework脚本对Open vSwitch工具命令的一些封装））

当网络中的机器足够多时会引发广播风暴，导致主机所在网络性能的下降。同时，不同部门、不同组织的机器连在同一个二层网络中也会造成安全问题。因此，在交换机中划分子网、隔离广播域的思路便形成了VLAN的概念。VLAN技术将一个二层网络的机器隔离开来，那么如何区分不同VLAN的流量呢？IEEE802.1q协议规定了VLAN的实现方法，在传统的以太网帧（传统的以太网是没有VLAN tag字段的）中再添加一个VLAN tag字段，用于标示不同的VLAN。这样，支持VLAN的交换机在转发帧时，不仅会关注MAC地址，还会考虑到VLAN tag字段。VLAN tag中包含了TPIDPCP、CFI、VID，其中VID（VLAN ID）部分用来具体指出帧时属于哪个VLAN的。VID占12位，所以其取值范围为0到4095。

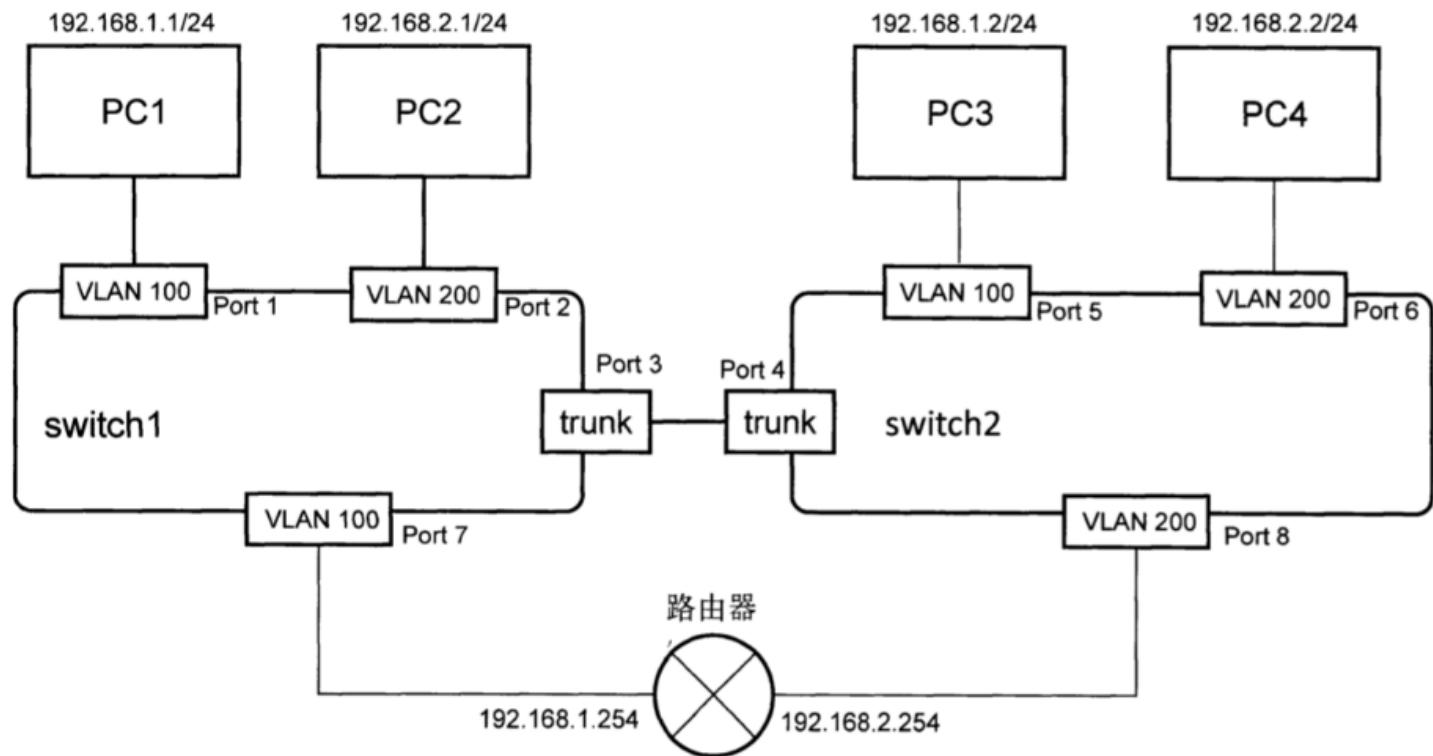


图4-5 多交换机VLAN划分

介绍一下交换机的access端口和trunk端口。图中，

- Port1、Port2、Port5、Port6、Port7、Port8为access端口，每一个access端口都会分配一个VLAN ID，标示它所连接的设备属于哪一个VLAN。

当数据帧从外界通过access端口进入交换机时，数据帧原来是不带tag的，access端口给数据帧打上

tag (VLAN ID即为access端口所分配的VLAN ID)，有tag就不能更改了，也不再重新打了；当数据帧从交换机内部通过access端口发送时，数据帧的VLAN ID必须和access端口的VLAN ID一致，access端口接收此帧，接着access端口将帧的tag信息去掉，再发送出去。

- Port3、Port4为trunk端口，trunk端口不属于某个特定的VLAN，而是交换机和交换机之间多个VLAN的通道。trunk端口声明了一组VLAN ID，表明只允许带有这些VLAN ID的数据帧通过，从trunk端口进入和出去的数据帧都是带tag的（不考虑默认VLAN的情况）。PC1和PC3属于VLAN100，PC2和PC4属于VLAN200，所以PC1和PC3处于同一个二层网络中，PC2和PC4处在同一个二层网络中。尽管PC1和PC2连接在同一台交换机中，但它之间的通信是需要经过路由器的。

VLAN tag是如何发挥作用的呢？当PC1向PC3发送数据时，PC1将IP包封装在以太帧中，帧的目的MAC地址为PC3的地址，此时帧并没有tag信息。当帧到达Port1时，Port1给帧打上tag (VID=100)，帧进入switch1，然后帧通过Port3、Port4到达Switch2 (Port3、Port4允许VLAN ID为100/200的帧通过)。在switch2中，Port5所标记的VID和帧相同，MAC地址也匹配，帧就发送到Port5上，Port5将帧的tag信息去掉，然后发给PC3。由于PC2、PC4与PC1的VLAN不同，因此收不到PC1发出的帧。

4.1 使用OVS对单主机上Docker容器的VLAN划分（最后是同主机且都在一个子网范围，A、C容器能互通，B、D容器能互通）

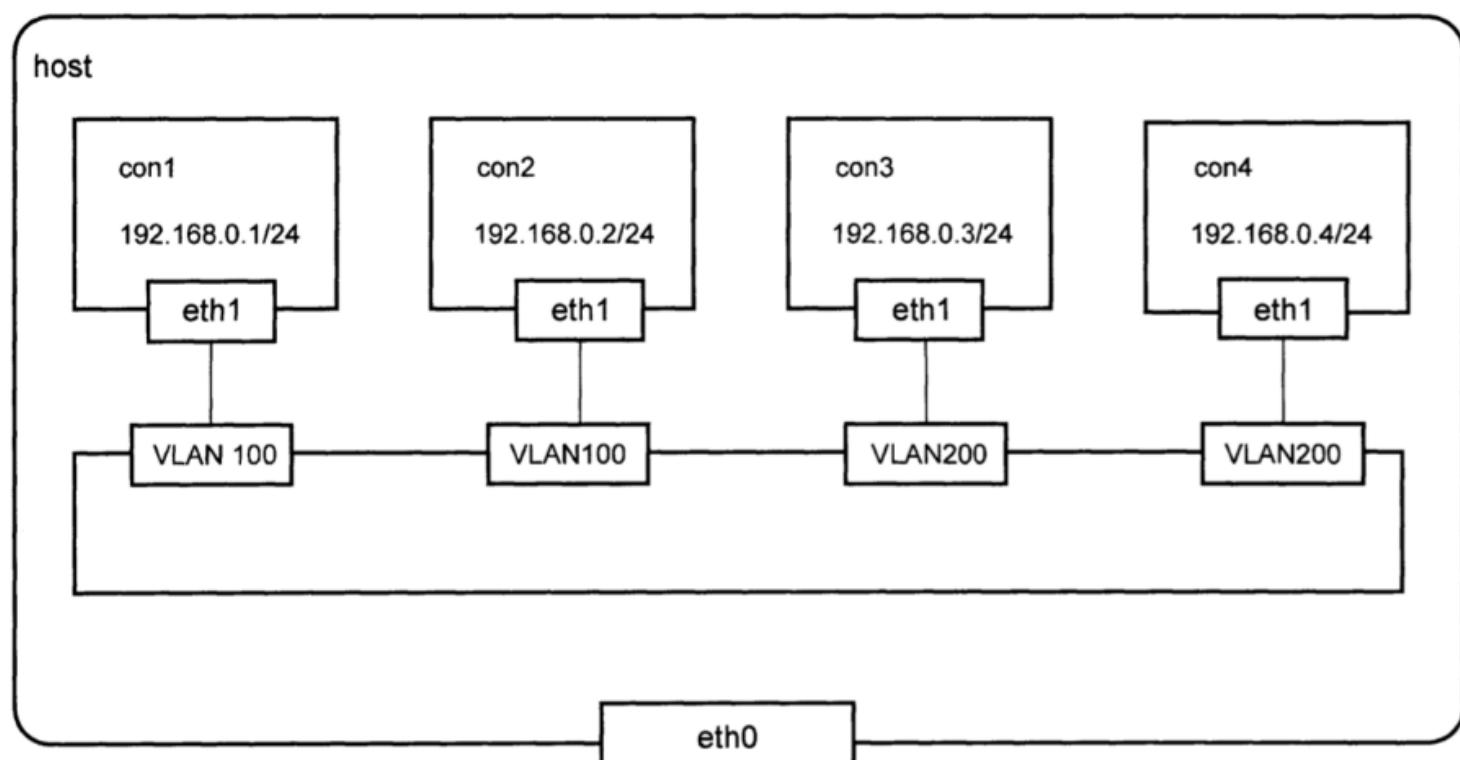


图4-6 单主机Docker容器VLAN划分

```
1. #在主机A上创建4个Docker容器: con1、con2、con3、con4
2. $ docker run -itd --name con1 ubuntu /bin/bash
3. $ docker run -itd --name con2 ubuntu /bin/bash
4. $ docker run -itd --name con3 ubuntu /bin/bash
5. $ docker run -itd --name con4 ubuntu /bin/bash
6.
7. #使用pipework将con1、con2划分到一个VLAN中，使用ovs0网桥，不存在则创建
8. $ pipework ovs0 con1 192.168.0.1/24 @100
9. $ pipework ovs0 con2 192.168.0.2/24 @100
10.
11. #使用pipework将con3、con4划分到一个VLAN中
12. $ pipework ovs0 con3 192.168.0.3/24 @200
13. $ pipework ovs0 con4 192.168.0.4/24 @200
```

pipework配置完成后，每个容器都多了一块eth1网卡，eth1连在ovs0网桥上，并且进行了VLAN的隔离。和之前一样，通过nc命令测试各容器之间的连通性时发现，con1和con2可以相互通信，但与con3和con4隔离。如此一来，一个简单的VLAN隔离容器网络就完成了。

使用Open vSwitch配置VLAN比较简单，如创建access端口和trunk端口使用如下命令：

```
1. #在ovs0网桥上增加两个端口port1、port2
2. $ ovs-vsctl add-port ovs0 port1 tag=100
3. $ ovs-vsctl add-port ovs0 port2 trunk=100,200
```

在向Open vSwitch中添加端口时，若不添加任何限制，此端口则转发所有帧。

4.2 使用OVS对多主机Docker容器的VLAN划分 leengine的最终方案

多主机VLAN的情况下，肯定有属于同一VLAN但又在不同主机上的容器，因此多主机VLAN划分的前提是跨主机通信。

要使不同主机上的容器处于同一VLAN，就只能采用桥接方式。首先用桥接的方式将所有容器连接在一个逻辑交换机上，再根据情况进行VLAN的划分。桥接需要将主机的一块网卡桥接到容器所连接的Open vSwitch网桥上（桥接要求必须两个网卡或以上，eth0网卡连外网，eth1网卡连接docker0网桥），这就需要一块额外的网卡eth1来完成，桥接的网卡需要开启混杂模式。

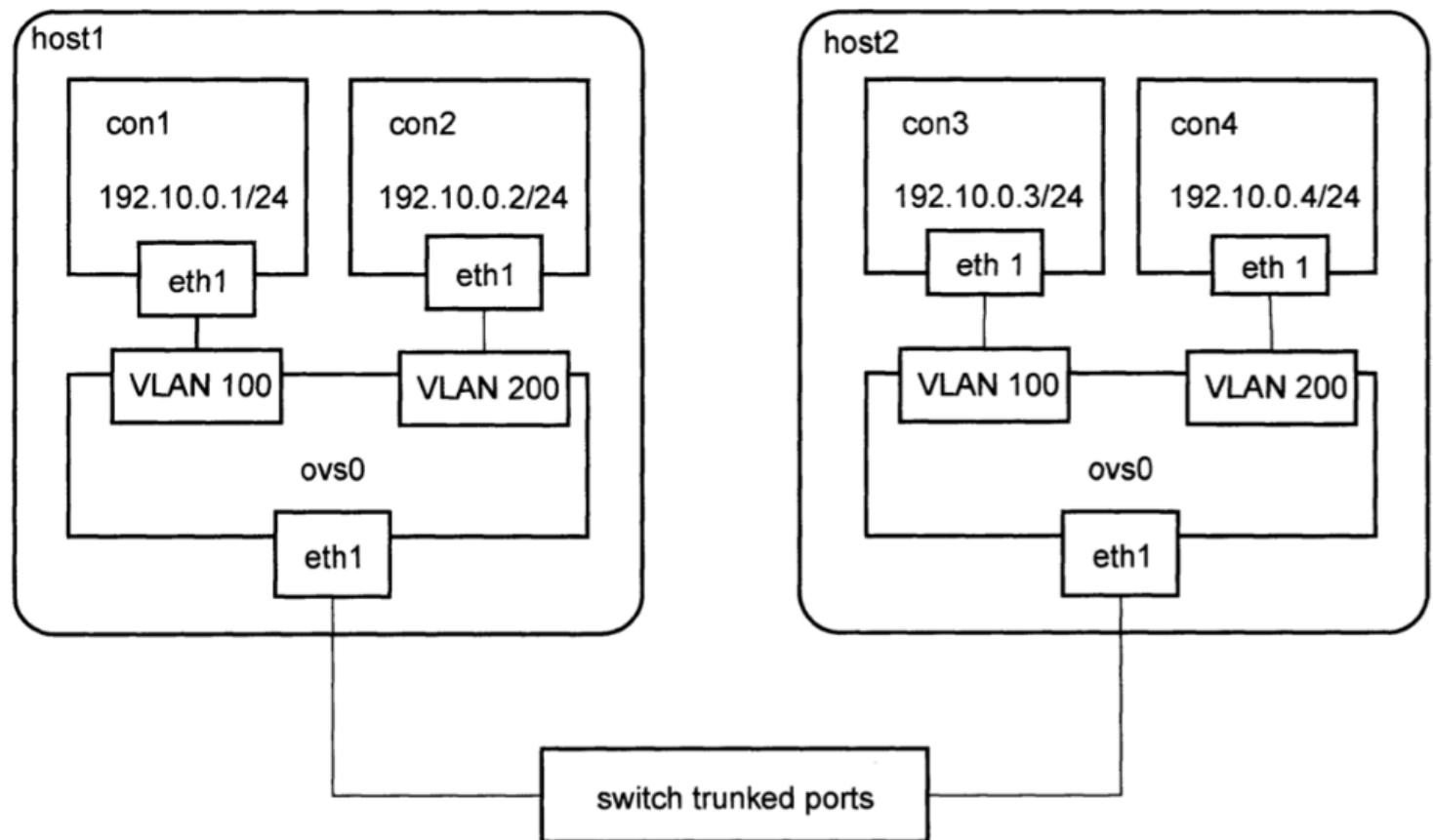


图4-7 多主机Docker容器VLAN划分

如上图中，host1上的con1和host2上的con3属于VLAN100，con2和con4属于VLAN200.由于会有VLAN ID为100和VLAN ID为200的帧通过，物理交换机上连接host1和host2的端口应设置为trunk端口。host1和host2上eth1没有设置VLAN的限制（trunk），是允许所有帧通过的。完成上图例子需要作如下操作：

```
1. #在host1上
2. $ docker run -itd --name con1 ubuntu /bin/bash
3. $ docker run -itd --name con2 ubuntu /bin/bash
4. $ pipework ovs0 con1 192.168.0.1/24 @100
5. $ pipework ovs0 con1 192.168.0.2/24 @200
6. $ ovs-vsctl add-port ovs0 eth1;
```

```
7.  
8. #在host2上  
9. $ docker run -itd --name con3 ubuntu /bin/bash  
10. $ docker run -itd --name con4 ubuntu /bin/bash  
11. $ pipework ovs0 con3 192.168.0.3/24 @100  
12. $ pipework ovs0 con4 192.168.0.4/24 @200  
13. #此端口则转发所有帧  
14. $ ovs-vsctl add-port ovs0 eth1;
```

5 OVS隧道模式

5.1 Overlay技术模型

当前主要的Overlay技术有 **VXLAN** (Virtual Extensible LAN) 和 **NVGRE** (Network Virtualization using Generic Routing Encapsulation)。

- **VXLAN**是将以太网报文封装在UDP传输层上的一种隧道转发模式，它采用24位比特表示二层网络分段，称为VNI (VXLAN Network Identifier)，类似于VLAN ID的作用。
- **NVGRE**同VXLAN类似，它使用GRE的方法来打通二层与三层之间的通路，采用24位比特的GRE key来作为网络标识 (TNI)

5.2 GRE简介

5.3 GRE实现Docker容器跨网络通信（容器在同一个子网中）

5.4 GRE实现Docker容器跨网络通信（容器在不同子网中）

5.5 多租户环境下的GRE网络

6 隧道方案Flannel Overlay网络

7 隧道方案Weave Overlay网络

8 MacVlan+VLAN构建网络

9 路由方案Calico网络

10 Docker1.9后内置Overlay网络（对比所有的Overlay技术）

二层VLAN网络的解决跨主机通信的思路是把原先的网络架构改造为互通的大二层网络，通过特定网络设备直接路由，实现容器点到点的之间通信。这种方案在传输效率上比Overlay网络占优，然而它也存在一些固有的问题。

1. 这种方法需要二层网络设备支持，通用性和灵活性不如后者；
2. 由于通常交换机可用的VLAN数量都在4000个左右，这会对容器集群规模造成限制，远远不能满足公有云或大型私有云的部署需求；
3. 大型数据中心部署VLAN，会导致任何一个VLAN的广播数据会在整个数据中心内泛滥，大量消耗网络带宽，带来维护的困难。

相比之下，Overlay网络是指在不改变现有网络基础设施的前提下，通过某种约定通信协议，把二层报文封装在IP报文之上的新的数据格式。这样不但能够充分利用成熟的IP路由协议进程数据分发，而且在Overlay技术中采用扩展的隔离标识位数，能够突破VLAN的4000数量限制，支持高达16M的用户，并在必要时可将广播流量转化为组播流量，避免广播数据泛滥。因此，Overlay网络实际上是目前最主流的容器跨节点数据传输和路由方案。

很多实现CNM的第三方，如Weave、Flannel、Calico，docker1.9后支持创建容器指定--net=overlay，不过需要配合docker swarm使用。

11 如何使用DNS、域名方式让访问服务

Kubernetes

要立即开

容器网络、跨主机网络

要立即开

四 In Arch

关系、非关系、时序数据库/mycat/数据库水平、垂直拆分

mysql

B树、B-树、B+树

B树：

二叉树；

所有非叶子结点至多拥有两个儿子；

每个结点只存储一个关键字；

非叶子结点的左指针指向小于其关键字的子树，右指针指向大于其关键字的子树。

搜索时，等于则命中，小于走左结点，大于走右结点。

B-树：

是一种多路搜索树；

任意非叶子结点最多只有M个儿子， $M > 2$ ；

根结点的儿子数为 $[2, M]$ ；

除根结点以外的非叶子结点的儿子数为 $[M/2, M]$ ；

每个结点存放至少 $M/2-1$ （取上整）和至多 $M-1$ 个关键字；（至少2个关键字）；

非叶子结点的关键字个数=指向儿子的指针个数-1；

非叶子结点的关键字： $K[1], K[2], \dots, K[M-1]$ ；且 $K[i] < K[i+1]$ ；

非叶子结点的指针： $P[1], P[2], \dots, P[M]$ ；其中 $P[1]$ 指向关键字小于 $K[1]$ 的子树， $P[M]$ 指向关键字大于 $K[M-1]$ 的子树，其它 $P[i]$ 指向关键字属于 $(K[i-1], K[i])$ 的子树；

所有叶子结点位于同一层；

所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：

除B-树特点外，还有：

非叶子结点的子树指针与关键字个数相同；

非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

为所有叶子结点增加一个链指针；

所有关键字都在叶子结点出现；

B+树总是到叶子结点才命中；

mysql中普遍使用B+Tree做索引，但在实现上又根据聚簇索引（Innodb存储引擎使用）和非聚簇索引（MyISAM存储引擎）不同，非聚簇索引比聚簇索引多了一次读取数据的IO操作，所以查找性能上会差。

Innodb引擎和MyIASM引擎对比

MyISAM引擎与InnoDB引擎相比较：

1. Innodb引擎提供了对数据库ACID事务的支持，并且实现了SQL标准的四种隔离级别，MyISAM没有提供对ACID事务的支持。

2. Innodb引擎提供了行级锁和外键约束，由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用Innodb引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表。MyISAM则当INSERT(插入)或UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。

3. MySQL运行时Innodb会在内存中建立缓冲池，用于缓冲数据和索引。

4. Innodb引擎不支持FULLTEXT类型的索引，所以不能做全文检索，而且没有保存表的行数，当SELECT COUNT(*) FROM TABLE时需要扫描全表。MyISAM都支持，而且保存了表的行数。

5. MyIASM相对简单，效率上要优于InnoDB，小型应用可以考虑使用MyIASM

6. MyIASM表保存成文件形式，跨平台使用更加方便

5.5之后，默认的引擎是InnoDB。

应用场景：

1. MyIASM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyIASM

2. InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB

一个很好的事务处理系统，必须具备这些标准特性：ACID（原子性、一致性、隔离性、持久性）

脏读、不可重复读、幻读概念：

脏读：当一个事务正在对数据进行修改，而这种修改还没有提交到数据库中，这时的另外一个事务也访问这个数据，然后使用了这个旧的数据。

不可重复读：我们两次读数据，在第二次读数据前，有个事务对数据做了更改，则我们第二次独到的数据可能与第一次不同。

幻读：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。

数据库隔离级别：

未提交读：事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读。

提交读：大多数数据库系统的默认隔离级别，但MySQL不是。一个事务开始时，只能“看见”已经提交的事务所做的修改。

可重复读：可重复读解决了脏读的问题，该隔离级别保证了在同一个事务中多次读取同样记录结果是一致的。但可重复读隔离级别还是无法解决幻读的问题。InnoDB和XtraDB存储引擎通过多版本并发控制(MVCC)解决了幻读的问题。TODO

可串行化：可串行化是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。可串行化会在读取每一行数据都加锁，所以可能导致大量的超时和锁争用问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

几种锁介绍：

读锁：也叫共享锁、S锁，若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

写锁：又称排他锁、X锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。这保证了其他事务在T释放A上的锁之前不能再读取和修改A。

表锁：操作对象是数据表。Mysql大多数锁策略都支持(innodb)，是系统开销最低但并发性最低的一个锁策略。事务T对整个表加读锁，则其他事务可读不可写，若加写锁，则其他事务增删改都不行。

行级锁：操作对象是数据表中的一行。是MVCC技术用的比较多的，但在MYISAM用不了。行级锁对系统开销较大，处理高并发较好。

MVCC：多版本并发控制。一般情况下，事务性储存引擎不是只使用表锁、行加锁的处理数据，而是结合了MVCC机制，以处理更多的并发问题。Mvcc处理高并发能力最强，但系统开销比最大（较表锁、行级锁），这是最求高并发付出的代价。

innodb MVCC主要是为可重复读事务隔离级别做的。在每一行记录的后面增加两个隐藏列，记录创建版本号和删除版本号，通过版本号来减少锁的争用。

Innodb MVCC的实现方式是：

事务以写锁的形式修改原始数据，把修改前的数据存放于undo log，通过回滚指针与主数据关联，修改成功(commit) 哪都不做，失败则恢复undo log中的数据(rollback)。

innodb 增删改查具体的执行过程：

UPDATE的事务过程：

begin->用写锁锁定该行->记录redo log->记录undo log->修改当前行的值，写事务编号，回滚指针指向undo log中的修改前的行

其实undo log分insert和update undo log，因为insert时，原始的数据并不存在，所以回滚时把insert undo log丢弃即可，而update undo log则必须遵守上述过程。

数据备份和恢复：

逻辑备份是将数据库的数据备份为一个文本文件，可以用mysqldump工具来完成逻辑备份。

逻辑备份的优点是对于各种存储引擎都可以用同样的方法来备份，更简单一些。

备份数据库test：mysqldump -uroot -p -l -F test > test.sql

恢复：mysqldump -uroot -p test < test.sql

物理备份，最大优点是备份和恢复的速度更快。因为都是直接基于文件的cp。屋里备份又分为冷备份和热备份两种。

冷备份，停掉数据库服务，直接cp数据文件的方法，对于MyISAM和InnoDB一样。

热备份，两个引擎方式不同，我们借助工具xtrabackup来完成热备，而且完成增量备份和全量备份，直接敲命令。

mysql5.6、5.7新特性

5.6 增加GTID

GTID事务是全局唯一性的，且一个事务对应一个GTID。GTID用来代替classic的复制方法，不在使用binlog+pos开启复制，对于传统三种复制后的又一新的复制方式。而是使用master_auto_positon=1的方式自动匹配GTID断点进行复制。MySQL-5.6.5开始支持的，MySQL-5.6.10后开始完善。

GTID的组成是：32位的带-的UUID-每台服务器上从1自增的序列。

5.7 增加并行复制和组提交

不懂。

主从复制

MySQL复制将主数据库的DDL和DML操作通过二进制日志传到从库上，然后在从库上对这些日志进行重做，从而使得从库和主库的数据保持同步。

MySQL复制的优点主要包括3个方面：

如果主库出现问题，可以快速切换到从库提供服务；

可以在从库上执行查询操作，降低主库的访问压力；

可以在从库上执行备份，以避免备份期间影响主库的服务。

MySQL的复制原理以及流程：

1.master将数据更新记录到二进制日志中；

2.从服务器上有个I/O线程，负责读取主服务器的二进制日志，将其保存为中继日志；还有个SQL线程，复制执行中继日志。从节点做复制几乎是实时进行的，但不是完全的实时，网络好的时候，而是异步的接近实时。

注意，备份是异步的复制，所以主从库存有一定的差距，在从库上进行的查询操作需要考虑到这些数据的差异，一般只有更新不频繁的数据或对实时性要求不高的数据可以通过从库查询，实时性要求高的数据仍然需要从主数据库获得。

MySQL复制方式

异步复制

MySQL默认的复制即是异步的，主库在执行完客户端提交的事务后会立即将结果返给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果挂掉了，此时主上已经提交的事务可能并没有传到从上，如果此时强行将从提升为主，可能导致新主上的数据不完整。

全同步复制

指当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。在异步复制中，master写数据到binlog且sync，slave request binlog后写入relay-log并flush disk。

半同步复制

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到中继日志中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，这个延迟最少是一个TCP/IP往返的时间。所以，半同步复制最好在低延时的网络中使用。

在全同步复制中，master写数据到binlog且sync，所有slave request binlog后写入relay-log并flush disk，并且回放完日志且commit。

mysql主备切换

使用mycat

mycat

数据拆分

数据的切分(Sharding)根据其切分规则的类型，可以分为两种切分模式。

一种是按照不同的表(或者 Schema)来切分到不同的数据库(主机)之上，这种切可以称之为数据的垂直(纵向)切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某个字段的某种规则拆分到多台数据库(主机)上面，这种切分称之为数据的水平(横向)切分。

垂直拆分缺点：

- 1.部分业务表无法 join，只能通过接口方式解决，提高了系统复杂度。
- 2.事务处理复杂。

由于垂直切分是按照业务的分类将表分散到不同的库，所以有些业务表会过于庞大，存在单库读写与存储瓶颈，所以就需要水平拆分来做解决。

水平拆分：

几种典型的分片规则包括：

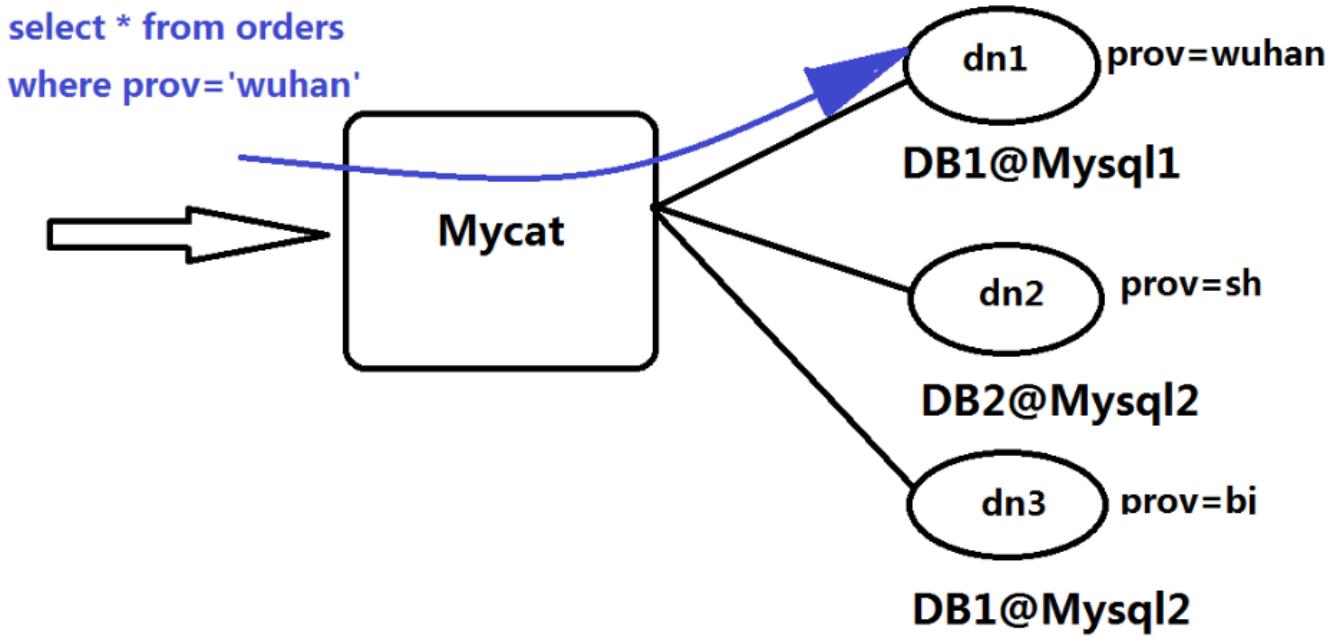
按照用户 ID 求模，将数据分散到不同的数据库，具有相同数据用户的都被分散到一个库中。

按照日期，将不同月甚至日的数据分散到不同的库中。

按照某个特定的字段求摸，或者根据特定范围段分散到不同的库中。

水平拆分缺点：

- 1.拆分规则难以抽象。
- 2.分片事务一致性难以解决。
- 3.跨库 join 性能较差。



例子：Orders表被分为三个分片datanode（简称dn），这三个分片是分布在两台mysql server(DataHost)上，即datanode=database@datahost方式，因此，你可以用一台到N台服务器来分片，分片规则为典型的字符串枚举分片规则。

一个规则的定义是分片字段+分片函数，这里的分片字段为prov，而分片函数为字符串枚举方式。

当Mycat收到一个SQL时，先会解析这个SQL，查找涉及到的表，然后看此表的定义，如果有分片规则，则获取到SQL里分片字段的值，并匹配分片函数，得到该SQL对应的分片列表，然后将SQL发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。

以 select * from Orders where prov=?语句为例，查到prov=wuhan，按照分片函数，wuhan返回dn1，于是SQL就发给了MySQL1，去取DB1上的查询结果，并返回给用户。

上述SQL改为 select * from Orders where prov in ('wuhan', 'beijing')，那么，SQL就会发给MySQL1与MySQL2去执行，然后结果集合并后输出给用户。但通常业务中我们的SQL会有Order By以及Limit 翻页语法，此时就涉及到结果集在Mycat端的二次处理，这部分的代码也比较复杂，而最复杂的则属两个表的Join问题，为此，Mycat提出了创新性的ER分片、全局表、Catlet、以及结合 Storm/Spark 引擎等十八般武艺的解决办法，从而成为目前业界最强大的方案，这就是开源的力量！

MyCat应用场景

- 单纯的读写分离，此时配置最为简单，支持读写分离，主从切换
- 分表分库，对于超过1000万的表进行分片，最大支持1000亿的单表分片
- 作为海量数据实时查询的一种简单有效方案，比如 100 亿条频繁查询的记录需要在3 秒内查询出来结果，除了基于主键的查询，还可能存在范围查询或其他属性查询，此时 Mycat 可能是最简单有效的选择

Mycat 目前有哪些功能与特性？

- 支持galera formysql集群，percona-cluster或者mariadb cluster，提供高可用性数据分片集群
- 支持NIO与AIO两种网络通信机制，Windows下建议AIO，Linux下目前建议NIO

Mycat 支持批量插入吗？

目前 Mycat1.3.0.3 以后支持多 values 的批量插入，如 insert into(xxx) values(xxx),(xxx)。

Mycat 支持多表 Join 吗？

Mycat 目前支持 2 个表 Join，后续会支持多表 Join。

Mycat 支持的或者不支持的语句有哪些？

insert into, 复杂子查询, 3 表及其以上跨库 join 等不支持。

MyCat目前不支持跨分片的事务，目前只是弱 XA 模式，还没完全实现 XA 模式。

Mycat基本概念

Mycat 是数据库中间件，提供了基本概念：

- 逻辑库(schema): 数据库中间件提供逻辑数据库概念，业务人员调用不关心是什么库，只要JDBC能连接上。
- 逻辑表(table): 读写数据的表就是逻辑表。逻辑表，可以是数据切分后，分布在一个或多个分片库中，也可以不做数据切分，不分片，只有一个表构成。
 - 分片表：那些原有的很大数据的表，需要切分到多个数据库的表，这样，每个分片都有一部分数据，所有分片构成了完整的数据。

例如在mycat配置中的t_node就属于分片表，数据按照规则被分到dn1,dn2两个分片节点(dataNode)上。

```
<table name="t_node" primaryKey="vid" autoIncrement="true" dataNode="dn1,dn2" rule="rule1" />
```

- 非分片表：一个数据库中并不是所有的表都很大，某些表是可以不用进行切分的，非分片是相对分片表来说的，就是那些不需要进行数据切分的表。

如下配置中t_node，只存在于分片节点(dataNode)dn1 上。

```
<table name="t_node" primaryKey="vid" autoIncrement="true" dataNode="dn1" />
```

- ER表：子表的记录与所关联的父表记录存放在同一个数据分片上，即子表依赖于父表，通过表分组(Table Group)保证数据Join不会跨库操作。

表分组(Table Group)是解决跨分片数据join的一种很好的思路，也是数据切分规划的重要一条规则。

- 全局表：对于大量的像字典表这类表，变动不频繁、数据量总体变化不大、数据规模不大，很少有超过数十万条记录。为解决业务表与这些附属的字典表之间的关联，就成了比较棘手的问题，mycat通过数据冗余来解决这类表的 join，即所有分片都有一份数据的拷贝，所有将字典表这类特性的表定义为全局表。

数据冗余是解决跨分片数据 join 的一种很好的思路，也是数据切分规划的另外一条重要规则。

- 分片节点(dataNode): 数据切分后，一个大表被分到不同的分片数据库上面，每个表分片所在的数据库就是分片节点 (dataNode)。
- 节点主机(dataHost): 数据切分后，每个分片节点(dataNode)不一定都会独占一台机器，同一机器上面可以有多个分片数据库，这样一个或多个分片节点(dataNode)所在的机器就是节点主机(dataHost)。
- 分片规则(rule): 数据切分，一个大表被分成若干个分片表，就需要一定的规则，这样按照某种业务规则把数据分到某个分片的规则就是分片规则。
-
- 全局序列号(sequence): 数据切分后，原有的关系数据库中的主键约束在分布式条件下将无法使用，因此需要引入外部机制保证数据唯一性标识，这种保证全局性的数据唯一标识的机制就是全局序列号(sequence)。

MyCat配置文件

MyCAT 目前主要通过配置文件的方式来定义逻辑库和相关配置：

- MYCAT_HOME/conf/schema.xml 中定义逻辑库，表、分片节点等内容。

- MYCAT_HOME/conf/rule.xml 中定义分片规则.
- MYCAT_HOME/conf/server.xml 中定义用户以及系统相关变量, 如端口等.

server.xml 定义信息

1. 可以配置SQL白名单和SQL黑名单。ip白名单用户对应的可以访问的ip地址。对于黑名单可以定义拦截规则，如是否允许修改sql、删除sql、查询sql等。

```

1. <firewall>
2.   <whitehost>
3.     <host user="mycat" host="127.0.0.1"></host> <!--ip 白名单 用户对应的可以访问的 ip 地址-->
4.   </whitehost>
5.   <blacklist check="true">
6.     <property name="selectAllow">false</property> <!!--黑名单允许的 权限 后面为默认-->
7.   </blacklist>
8. </firewall>
```

2. server.xml中标签：

schema标签用于定义mycat实例中的逻辑库。mycat可以定义多个逻辑库，每个库都有自己的配置。

table标签用于定义mycat受理中的逻辑库，所有需要拆分的表都需要在这个标签中定义。

childTable标签用于定义E-R分片的子表。通过标签上的属性与父表进行关联。

dataNode标签定义了MyCat中的数据节点，也就是我们通常说所的数据分片。

dataHost标签直接定义了具体的数据库实例、读写分离配置和心跳语句。

```

1. <schema name="TESTDB" checkSQLSchema="false" sqlMaxLimit="100">
2.   <table name="travelrecord" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" ></table>
3. </schema>
4. <schema name="USERDB" checkSQLSchema="false" sqlMaxLimit="100">
5.   <table name="company" dataNode="dn10,dn11,dn12" rule="auto-sharding-long" ></table>
6. </schema>
7. <dataNode name="dn1" dataHost="lch3307" database="db1" ></dataNode>
8. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="0" writeType="0" dbType="m
9.   <heartbeat>select user()</heartbeat>
10.  <!-- can have multi write hosts -->
11.  <writeHost host="hostM1" url="localhost:3306" user="root" password="123456">
12.  <!-- can have multi read hosts -->
13.  <!-- <readHost host="hostS1" url="localhost:3306" user="root" password="123456"/> -->
14.  </writeHost>
15.  <!-- <writeHost host="hostM2" url="localhost:3316" user="root" password="123456"/> -->
16. </dataHost>
17. <!--
18. dataHost属性介绍:
19. name属性      唯一标识 dataHost 标签, 供上层的标签使用。
20. maxCon属性    指定每个读写实例连接池的最大连接。
21. ...
22. balance属性   负载均衡类型, 目前的取值有3种:
23. 1. balance="0", 不开启读写分离机制, 所有读操作都发送到当前可用的writeHost上。
24. 2. balance="1", 全部的readHost与stand by writeHost参与select语句的负载均衡, 简单的说, 当双
25. 主双从模式(M1->S1, M2->S2, 并且 M1 与 M2 互为主备), 正常情况下, M2,S1,S2 都参与select语句的负载均衡。
26. 3. balance="2", 所有读操作都随机的在 writeHost、readhost 上分发。
27. 4. balance="3", 所有读请求随机的分发到 writerHost 对应的 readhost 执行, writerHost 不负担读压 力
28. writeType属性  负载均衡类型, 目前的取值有3种:
29. 1. writeType="0", 所有写操作发送到配置的第一个 writeHost, 第一个挂了切到还生存的第二个 writeHost, 重
30. 2. writeType="1", 所有写操作都随机的发送到配置的 writeHost, 1.5 以后废弃不推荐。
31. switchType属性
32. -1 表示不自动切换
33. 1 默认值, 自动切换
34. 2 基于MySQL主从同步的状态决定是否切换 心跳语句为 show slave status
```

```
35. 3 基于MySQL galaxy cluster的切换机制(适合集群)(1.4.1) 心跳语句为 show status like 'wsrep%'.
36. heartbeat标签 指明用于和后端数据库进行心跳检查的语句。
37. writeHost标签、readHost标签 指定后端数据库的相关配置给 mycat，用于实例化后端连接池。
38. -->
```

如上所示的配置就配置了两个不同的逻辑库，逻辑库的概念和 MySQL 数据库中 Database 的概念相同。

server.xml 定义信息保存了所有 mycat 需要的系统配置信息。

user标签 定义了登录mycat的用户和权限。例如上面的例子中，我定义了一个用户，用户名为test、密码也为test，可访问的schema也只有TESTDB一个。

如果我在schema.xml中定义了多个schema，那么这个用户是无法访问其他的schema。在mysql客户端看来则是无法使用use切换到这个其他的数据库。

privileges子节点 对用户的schema及下级的table进行精细化的DML权限控制

```
1. <user name="zhuam">
2.   <property name="password">111111</property>
3.   <property name="schemas">TESTDB,TESTDB1</property>
4.   <!-- 表级权限：Table 级的 dml(curd) 控制，未设置的 Table 继承 schema 的 dml -->
5.   <!-- TODO：非 CURD SQL 语句，透明传递至后端 -->
6.   <privileges check="true">
7.     <schema name="TESTDB" dml="0110" >
8.       <table name="table01" dml="0111"></table>
9.       <table name="table02" dml="1111"></table>
10.      </schema>
11.      <schema name="TESTDB1" dml="0110" >
12.        <table name="table03" dml="1110"></table>
13.        <table name="table04" dml="1010"></table>
14.      </schema>
15.    </privileges>
16.  </user>
```

system标签 这个标签内嵌套的所有 property 标签都与系统配置有关。

rule.xml 定义信息里面就定义了我们对表进行拆分所涉及到的规则定义。我们可以灵活的对表使用不同的分片算法。

tableRule标签 定义表规则。

```
1. <tableRule name="rule1">
2.   <rule>
3.     <columns>id</columns>
4.     <algorithm>func1</algorithm>
5.   </rule>
6. </tableRule>
7. <!--
8. name 属性 指定唯一的名字，用于标识不同的表规则。
9. 内嵌的rule标签
10. columns 指定要拆分的列名字。
11. algorithm 使用function标签中的name属性。连接表规则和具体路由算法。当然，多个表规则可以连接到同一个路由！
-->
13. function 标签 指定拆分算法
14. <function name="hash-int" class="org.opencloudroute.function.PartitionByFileMap">
15.   <property name="mapFile">partition-hash-int.txt</property>
16. </function>
17. <!--
18. name 指定算法的名字。
```

19. `class` 制定路由算法具体的类名字。
20. `property` 为具体算法需要用到的一些属性。
21. -->

MyCat跨分片join

对于多表的join, Mycat目前版本支持跨分片的join, 主要实现的方式有四种。

全局表, ER分片, catletT(人工智能)和ShareJoin, ShareJoin在开发版中支持, 前面三种方式1.3.0.1支持。全局表和ER分片解决了80%以上的企业应用所面临的问题。

全局表

一个真实的业务系统中, 往往存在大量的类似字典表的表格, 考虑到字典表具有以下几个特性:

- 变动不频繁
- 数据量总体变化不大
- 数据规模不大, 很少有超过数十万条记录。

鉴于此, MyCAT 定义了一种特殊的表, 称之为“全局表”, 全局表具有以下特性:

- 全局表的插入、更新操作会实时在所有节点上执行, 保持各个分片的数据一致性
- 全局表的查询操作, 只从一个节点获取
- 全局表可以跟任何一个表进行JOIN操作

将字典表或者符合字典表特性的一些表定义为全局表。

全局表配置比较简单, 不用写Rule规则, 如下配置即可:

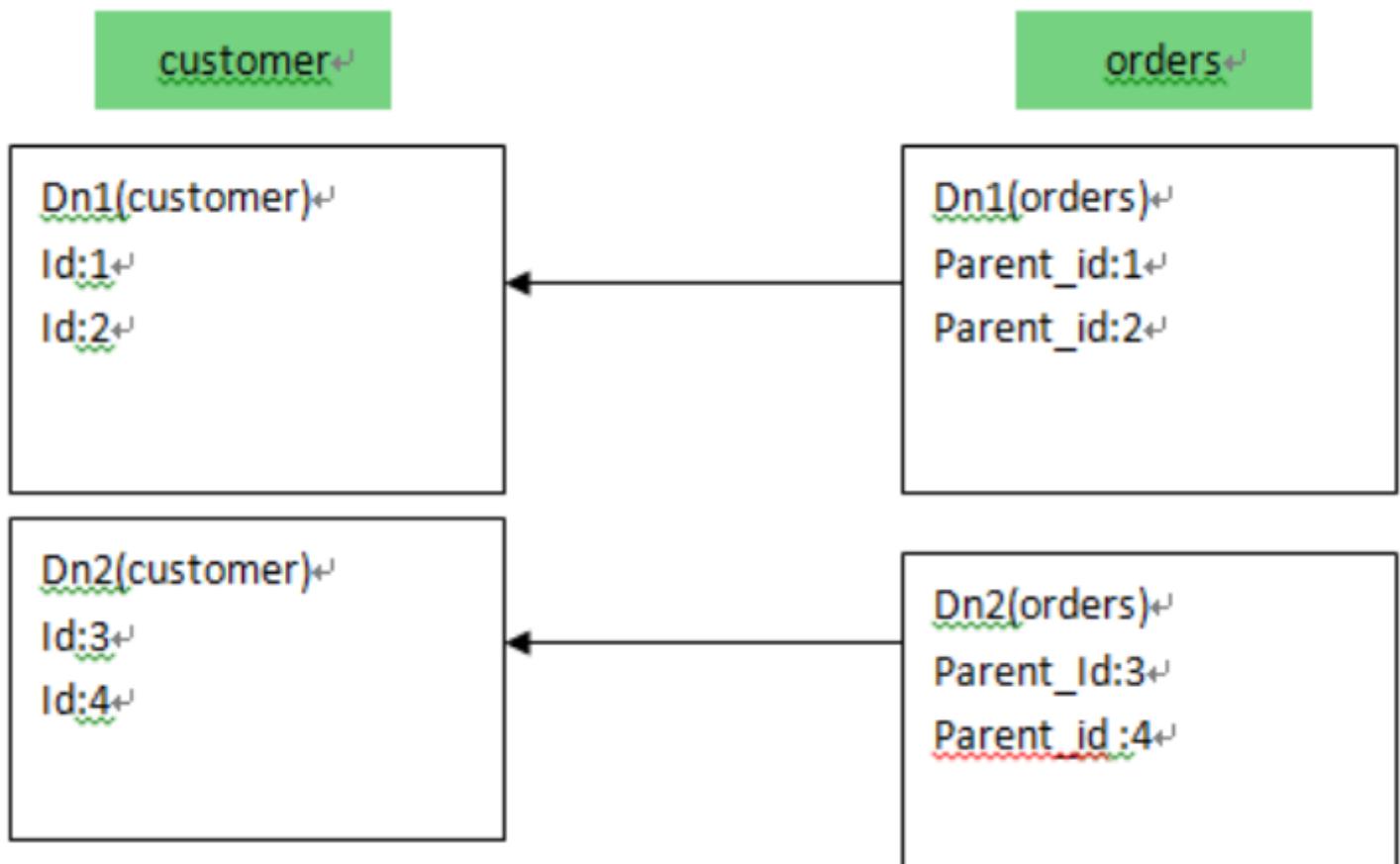
```
1. <table name="company" primaryKey="ID" type="global" dataNode="dn1,dn2,dn3" />
```

需要注意的是, 全局表每个分片节点上都要有运行创建表的 DDL 语句。

ER Join

子表的记录与所关联的父表记录存放在同一个数据分片上。

customer采用sharding-by-intfile这个分片策略, 分片在dn1,dn2上, orders依赖父表进行分片, 两个表的关联关系为orders.customer_id=customer.id。



这样一来，分片Dn1上的的customer与Dn1上的orders就可以进行局部的JOIN联合，Dn2上也如此，再合并两个节点的数据即可完成整体的JOIN，试想一下，每个分片上 orders 表有 100 万条，则 10 个分片就有 1 个亿，基于 E-R 映射的数据分片模式.

schema.xml 中定义如下的分片配置：

```

1. <table name="customer" dataNode="dn1,dn2" rule="sharding-by-intfile">
2.   <childTable name="orders" joinKey="customer_id" parentKey="id"/>
3. </table>

```

其他方式不讲了，都是很不固定的方法，我们项目中应该做到前面两种。

全局序列号

在实现分库分表的情况下，数据库自增主键已无法保证自增主键的全局唯一。为此，MyCat 提供了全局 sequence，并且提供了包含本地配置、数据库配置、本地时间戳、分布式ZK ID生成器、ZK递增方式实现方式。

本地文件方式：此方式MyCAT将sequence配置到文件中，当使用到sequence中的配置后，MyCAT会到classpath中的sequence_conf.properties文件中 sequence获取当前的值。

server.xml 中配置：

```

1. <system>
2. <property name="sequnceHandlerType">0</property>
3. </system>

```

数据库方式：在数据库创建一张表，包含序列名称，当前值，自增值，用来存放sequence，创建相关函数，然后配置sequnceHandlerType的值为1.

还有。

不管哪种方式，使用方式都是：

```
1. insert into table1(id,name) values(next value for MYCATSEQ_GLOBAL,' test' );
```

我们用的数据库方式。

mycat对于主键定义成自增类型也提供了last_insert_id()返回主键值。

mycat定义逻辑表时可指定主键字段是哪个，是否自增。

使用mybatis获得自增主键方法：

```
1. <insert id="insert" parameterType="Person">
2.     <selectKey keyProperty="id" resultType="long">
3.         select LAST_INSERT_ID()
4.     </selectKey>
5.     insert into person(name,pswd) values(#{"name"},#{pswd})
6. </insert>
```

分片规则

在数据切分处理中，特别是水平切分中，中间件最终要的两个处理过程就是数据的切分、数据的聚合。选择合适的切分规则，至关重要，因为它决定了后续数据聚合的难易程度，甚至可以避免跨库的数据聚合处理。前面讲了数据切分中重要的几条原则，其中有几条是数据冗余，表分组（Table Group），这都是业务上规避跨库 join 的很好的方式，但不是所有的业务场景都适合这样的规则，因此本章将讲述如何选择合适的切分规则。

切分暂解决不了问题：

有一类业务场景是“主表 A+关系表+主表 B”，举例来说就是商户会员+订单+商户，对应这类业务，如何切分？从会员的角度，如果需要查询会员购买的订单，那按照会员进行切分即可，但是如果要查询商户当天售出的订单，那又需要按照商户做切分，可是如果既要按照会员又要按照商户切分，几乎是无法实现，这类业务如何选择切分规则非常难。目前还暂时无法很好支持这种模式下的 3 个表之间的关联。目前总的原则是需要从业务角度来看，关系表更偏向哪个表，即“A 的关系”还是“B 的关系”，来决定关系表跟从那个方向存储，未来 Mycat 版本中将考虑将中间表进行双向复制，以实现从 A-关系表 以及 B-关系表的双向关联查询。

主键分片 vs 非主键分片

当没有任何字段可以作为分片字段的时候，主键分片就是唯一选择，其优点是按照主键的查询最快，当采用自动增长的序列号作为主键时，还能比较均匀的将数据分片在不同的节点上。若有某个合适的业务字段比较合适作为分片字段，则建议采用此业务字段分片，选择分片字段的条件如下：

- 尽可能的比较均匀分布数据到各个节点上；
- 该业务字段是最频繁的或者最重要的查询条件。

当你找到某个合适的业务字段作为分片字段以后，不必纠结于“牺牲了按主键查询记录的性能”，因为在这种情况下，MyCAT 提供了“主键到分片”的内存缓存机制，热点数据按照主键查询，丝毫不损失性能。

```
1. <table name="t_user" primaryKey="user_id" dataNode="dn$1-32" rule="mod-long">
2.   <childTable name="t_user_detail" primaryKey="id" joinKey="user_id" parentKey="user_id"
3. </table>
```

Mycat 常用的分片规则:

1. 分片枚举

本规则适合于配置固定数据项的业务，如省份/区县。配置如下：

```
1. <tableRule name="sharding-by-intfile">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>hash-int</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="hash-int" class="org.opencloudb.route.function.PartitionByFileMap">
8.   <property name="mapFile">partition-hash-int.txt</property>
9.   <property name="type">0</property>
10.  <property name="defaultNode">0</property>
11. </function>
```

partition-hash-int.txt 配置：

```
1. 10000=0
2. 10010=1
3. DEFAULT_NODE=1
```

配置说明：

columns 标识将要分片的表字段

algorithm 分片函数，

其中分片函数配置中，

mapFile 标识配置文件名称，

type 默认值为 0，0 表示 Integer，非零表示 String，所有的节点配置都是从 0 开始，0 代表的是配置中第 1 个配置。

defaultNode 表示默认节点，小于 0 表示不设置默认节点，大于等于 0 表示设置默认节点。枚举分片时，如果碰到不识别的枚举值，就让它路由到默认节点，如果不配置默认节点（defaultNode 值小于 0 表示不配置默认节点），碰到不识别的枚举值就会报错。

2. 固定分片 hash 算法

本条规则是二进制的操作，取 id 的二进制低 10 位，即 id 二进制 &1111111111。此算法的优点在于如果按照 10 进制取模运算，在连续插入 1-10 时候 1-10 会被分到 1-10 个分片，增大了插入的事务控制难度，而此算法根据二进制则可能会分到连续的分片，减少插入事务控制难度。

```
1. <tableRule name="rule1">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>func1</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="func1" class="org.opencloudb.route.function.PartitionByLong">
8.   <property name="partitionCount">2,1</property>
```

```
9.     <property name="partitionLength">256,512</property>
10.    </function>
```

配置说明：

columns 标识将要分片的表字段

algorithm 分片函数，

partitionCount 分片个数列表，

partitionLength 分片范围列表

分区长度：默认为最大 $2^n=1024$ ，即最大支持 1024

上面例子解释一下，希望将数据水平分成 3 份，前两份各占 25%，第三份占 50%。

```
1. // |<-----1024----->|
2. // |<----256---->|<----256---->|<----512---->|
3. // |      partition0      |      partition1      |      partition2      |
4. // |          共 2 份,故 count[0]=2           |共1份,故count[1]=1   |
5. int[] count = new int[] { 2, 1 };
6. int[] length = new int[] { 256, 512 };
7. PartitionUtil pu = new PartitionUtil(count, length);
```

如果需要平均分配设置：平均分为 4 分片，partitionCount*partitionLength=1024

```
1. <function name="func1" class="org.opencloudb.route.function.PartitionByLong">
2.     <property name="partitionCount">4</property>
3.     <property name="partitionLength">256</property>
4. </function>
```

3. 范围约定

此分片适用于，提前规划好分片字段某个范围属于哪个分片，

start \leq range \leq end.

K=1000, M=10000.

```
1. <tableRule name="auto-sharding-long">
2.     <rule>
3.         <columns>user_id</columns>
4.         <algorithm>rang-long</algorithm>
5.     </rule>
6. </tableRule>
7. <function name="rang-long" class="org.opencloudb.route.function.AutoPartitionByLong">
8.     <property name="mapFile">autopartition-long.txt</property>
9.     <property name="defaultNode">0</property>
10.    </function>
```

配置说明：

columns 标识将要分片的表字段，

algorithm 分片函数，

rang-long 函数中

mapFile 代表配置文件路径

defaultNode 超过范围后的默认节点。

所有的节点配置都是从 0 开始，0 代表的是配置文件的第一条配置。

此配置非常简单，即预先制定可能的 id 范围到某个分片。

autopartition-long.txt 配置

```
1. 0-500M=0
2. 500M-1000M=1
3. 1000M-1500M=2
4. 或
5. 0-10000000=0
6. 10000001-20000000=1
```

4. 取模

此规则为对分片字段求模运算。

```
1. <tableRule name="mod-long">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>mod-long</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="mod-long" class="org.opencloudb.route.function.PartitionByMod">
8.   <!-- how many data nodes -->
9.   <property name="count">3</property>
10. </function>
```

配置说明：

columns标识将要分片的表字段，

algorithm分片函数，

此种配置根据 id 进行十进制求模，相比固定分片hash的二进制求模，在连续插入 1-10 时候 1-10 会被分到 1-10 个分片，增大了插入的事务控制难度.

5. 按日期（天）分片

此规则为按天分片。

```
1. <tableRule name="sharding-by-date">
2.   <rule>
3.     <columns>create_time</columns>
4.     <algorithm>sharding-by-date</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="sharding-by-date" class="org.opencloudb.route.function.PartitionByDate">
8.   <property name="dateFormat">yyyy-MM-dd</property>
9.   <property name="sBeginDate">2014-01-01</property>
10.  <property name="sEndDate">2014-01-02</property>
11.  <property name="sPartitionDay">10</property>
12. </function>
```

配置说明：

columns标识将要分片的表字段，

algorithm分片函数，

dateFormat日期格式

sBeginDate开始日期

sEndDate结束日期

sPartitionDay分区天数，即默认从开始日期算起，分隔 10 天一个分区

如果配置了sEndDate 则代表数据达到了这个日期的分片后循环从开始分片插入。

```
Assert.assertEquals(true, 0 == partition.calculate("2014-01-01"));
```

```
Assert.assertEquals(true, 0 == partition.calculate("2014-01-10"));
```

```
Assert.assertEquals(true, 1 == partition.calculate("2014-01-11"));
```

```
Assert.assertEquals(true, 12 == partition.calculate("2014-05-01"));
```

6. 取模范围约束

此种规则是取模运算与范围约束的结合，主要为了后续数据迁移做准备，即可以自主决定取模后数据的节点分布。

```
1. <tableRule name="sharding-by-pattern">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>sharding-by-pattern</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="sharding-by-pattern" class="org.opencloudb.route.function.PartitionByPatt
8.   <property name="patternValue">256</property>
9.   <property name="defaultNode">2</property>
10.  <property name="mapFile">partition-pattern.txt</property>
11. </function>
```

partition-pattern.txt

```
1. partition-pattern.txt
2. # id partition range start-end ,data node index
3. ##### first host configuration
4. 1-32=0
5. 33-64=1
6. 65-96=2
7. 97-128=3
8. ##### second host configuration
9. 129-160=4
10. 161-192=5
11. 193-224=6
12. 225-256=7
13. 0-0=7
```

配置说明：

columns标识将要分片的表字段，

algorithm分片函数，

patternValue即求模基数，

defaultNode默认节点，如果配置了默认，则不会按照求模运算

mapFile 配置文件路径

配置文件中，

1-32 即代表 $\text{id} \% 256$ 后分布的范围，如果在 1-32，则在分区 1，其他类推，如果 id 非数据，则会分配在 defaultNode 默认节点

```
1. String idVal = "0";
```

```
2.     Assert.assertEquals(true, 7 == autoPartition.calculate(idVal));  
3.     idVal = "45aaaaczxcz";  
4.     Assert.assertEquals(true, 2 == autoPartition.calculate(idVal));
```

7 截取数字做 hash 求模范围约束

此种规则类似于取模范围约束，此规则支持数据符号字母取模。

```
1. <tableRule name="sharding-by-prefixpattern">  
2.   <rule>  
3.     <columns>user_id</columns>  
4.     <algorithm>sharding-by-prefixpattern</algorithm>  
5.   </rule>  
6. </tableRule>  
7. <function name="sharding-by-pattern" class="org.opencloudb.route.function.PartitionByPref  
8.   <property name="patternValue">256</property>  
9.   <property name="prefixLength">5</property>  
10.  <property name="mapFile">partition-pattern.txt</property>  
11. </function>
```

partition-pattern.txt

```
1. partition-pattern.txt  
2. # range start-end ,data node index  
3. # ASCII  
4. # 8-57=0-9 阿拉伯数字  
5. # 64、65-90=@、A-Z  
6. # 97-122=a-z  
7. ##### first host configuration  
8. 1-4=0  
9. 5-8=1  
10. 9-12=2  
11. 13-16=3  
12. ##### second host configuration  
13. 17-20=4  
14. 21-24=5  
15. 25-28=6  
16. 29-32=7  
17. 0-0=7
```

配置说明：

columns 标识将要分片的表字段，

algorithm 分片函数，

patternValue 即求模基数，

prefixLength ASCII 截取的位数

mapFile 配置文件路径

配置文件中，

1-32 即代表 $\text{id} \% 256$ 后分布的范围，如果在 1-32 则在分区 1，其他类推，此种方式类似方式 6，只不过采取的是前 prefixLength 位列所有 ASCII 码的和进行求模

sum%patternValue，获取的值，在范围内的分片数，

```
1. String idVal="gf89f9a";
```

```
2.     Assert.assertEquals(true, 0==autoPartition.calculate(idVal));
3.     idVal="8df99a";
4.     Assert.assertEquals(true, 4==autoPartition.calculate(idVal));
5.     idVal="8dhdf99a";
6.     Assert.assertEquals(true, 3==autoPartition.calculate(idVal));
```

8. 应用指定

此规则是在运行阶段有应用自主决定路由到那个分片。

```
1. <tableRule name="sharding-by-substring">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>sharding-by-substring</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="sharding-by-substring" class="org.opencloud.route.function.PartitionDrie
8.   <property name="startIndex">0</property><!-- zero-based -->
9.   <property name="size">2</property>
10.  <property name="partitionCount">8</property>
11.  <property name="defaultPartition">0</property>
12. </function>
```

配置说明：

columns 标识将要分片的表字段，

algorithm 分片函数

此方法为直接根据字符串子串（必须是数字）计算分区号（由应用传递参数，显式指定分区号）。

例如 id=05-100000002

在此配置中代表根据 id 中从 startIndex=0, 开始，截取 size=2 位数字即 05, 05 就是获取的分区，如果没传默认分配到 defaultPartition。

如id=08-100000002,截取size=2位，即08。

9. 截取数字 hash 解析

此规则是截取字符串中的 int 数值 hash 分片。

```
1. <tableRule name="sharding-by-stringhash">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>sharding-by-stringhash</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="sharding-by-stringhash" class="org.opencloud.route.function.PartitionByS
8.   <property name="partitionLength">512</property><!-- zero-based -->
9.   <property name="partitionCount">2</property>
10.  <property name="hashSlice">0:2</property>
11. </function>
```

配置说明：

columns 标识将要分片的表字段，

algorithm 分片函数,

函数中

partitionLength 代表字符串 hash 求模基数,

partitionCount 分区数,

hashSlice hash 预算位, 即根据子字符串中 int 值 hash 运算

10. 一致性 hash

一致性 hash 预算有效解决了分布式数据的扩容问题。

```
1. <tableRule name="sharding-by-murmur">
2.   <rule>
3.     <columns>user_id</columns>
4.     <algorithm>murmur</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="murmur" class="org.opencloudb.route.function.PartitionByMurmurHash">
8.   <property name="seed">0</property><!-- 默认是 0-->
9.   <property name="count">2</property><!-- 要分片的数据库节点数量, 必须指定, 否则没法分片-->
10.  <property name="virtualBucketTimes">160</property><!-- 一个实际的数据库节点被映射为这么多虚
11.  <!--
12.  <property name="weightMapFile">weightMapFile</property>
13.  节点的权重, 没有指定权重的节点默认是 1。以 properties 文件的格式填写, 以从 0 开始到 count-1 的整数值也
14.  <!--
15.  <property name="bucketMapPath">/etc/mycat/bucketMapPath</property>
16.  用于测试时观察各物理节点与虚拟节点的分布情况, 如果指定了这个属性, 会把虚拟节点的 murmur hash 值与物理节点
17. </function>
```

11. 按单月小时拆分

此规则是单月内按照小时拆分, 最小粒度是小时, 可以一天最多 24 个分片, 最少 1 个分片, 一个月完后下月从头开始循环。

每个月月尾, 需要手工清理数据。

```
1. <tableRule name="sharding-by-hour">
2.   <rule>
3.     <columns>create_time</columns>
4.     <algorithm>sharding-by-hour</algorithm>
5.   </rule>
6. </tableRule>
7. <function name="sharding-by-hour" class="org.opencloudb.route.function.LatestMonthPartition"
8.   <property name="splitOneDay">24</property>
9. </function>
```

配置说明:

columns: 拆分字段, 字符串类型 (yyyymmddHH)

splitOneDay : 一天切分的分片数

12. 范围求模分片

13. 日期范围 hash 分片

14. 冷热数据分片

15. 自然月分片

按月份列分区，每个自然月一个分片，格式 between 操作解析的范例。

MyCat读写分离、主从模式下切换主从

有关读写分离分发规则的相关dataHost标签属性：

1. balance 负载均衡类型，目前取值有3种：

- (1) balance="0", 不开启读写分离机制，所有读操作都发送到当前可用的writeHost上。
- (2) balance="1", 全部的readHost与stand by writeHost 参与select语句负载均衡，简单的说，当双主双从模式(M1->S1, M2->S2, 并且M1与M2互为主备)，M2,S1,S2 都参与select语句的负载均衡。(类似于balance=3,读操作只分发到除了真正的主节点之外的所有节点)
- (3) balance="2", 所有读操作都随机的在 writeHost、readhost 上分发。
- (4) balance="3", 所有读请求随机的分发到 writeHost 对应的readhost执行，writeHost不负担读压力，注意balance=3 在 1.4 及其以后版本才有，1.3 没有。

balance值设置为1和设置成3效果应该是一样的。

有关主从(读写角色)切换设置属性规则的相关dataHost标签属性

1. switchType值：

- (1) -1 表示不自动切换
- (2) 1 默认值，自动切换
- (3) 2 基于mysql主从同步的状态决定是否切换
- (4) 3 基于cluster的切换，心跳语句要改成show status like 'wsrep%',这个里面配置的都是writehost

switchType值设置为1，表示自动切换，某些对主从数据一致要求较高的场景，建议使用2判断主从状态后再切换，或者使用galera cluster保证各节点数据一致性，将此值设置为3。

切换的触发条件为主节点mysql服务崩溃或停止。

2. slaveThreshold 主从的延迟在多少秒以内，则把读请求分发到这个从节点

有关写请求是否分发到多个写节点规则的相关dataHost标签属性

1. writeType值：

- (1) 0 表示只分发到当前的主节点
- (2) 1 表示分发到所有设定为writeHost的节点，不推荐使用，好像现在版本已经废除

若不想要自动切换功能，即 MySQL 写节点宕机后不自动切换到备用节点，则如下配置：

```
1. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="m
2.   <heartbeat>select user()</heartbeat>
3.   <!-- can have multi write hosts -->
4.   <writeHost host="hostM1" url="localhost:3306" user="root" password="123456">
5.     <!-- can have multi read hosts -->
6.     <readHost host="hostS1" url="localhost2:3306" user="root" password="123456" />
7.   </writeHost>
8. </dataHost>
```

如果要实现自动切换到备用节点，则如下配置：

```
1. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="m
2.   <heartbeat>select user()</heartbeat>
3.   <!-- can have multi write hosts -->
4.   <writeHost host="hostM1" url="localhost:3306" user="root" password="123456" />
5.   <writeHost host="hostS1" url="localhost2:3306" user="root" password="123456" />
6. </dataHost>
```

此时，第一个writeHost故障后，会自动切换到第二个，第二个故障后自动切换到第三个，当你是1主3从的模式的时候，可以把第一个从节点配置为writeHost2，第2个和第三个从节点则配置为 writeHost1的readHost，如下所示：

```
1. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="m
2.   <heartbeat>select user()</heartbeat>
3.   <writeHost host="hostM1" url="localhost:3306" user="root" password="123456" />
4.     <readHost host="hostS2" url="localhost3:3306" user="root" password="123456" />
5.     <readHost host="hostS3" url="localhost4:3306" user="root" password="123456" />
6.   </writeHost>
7.   <writeHost host="hostS1" url="localhost2:3306" user="root" password="123456" />
8. </dataHost>
```

如果读延时较大，使用根据主从延时的读写分离，或者强制走写节点。

应用强制走写：

1.6 以后添加了强制走读走写处理：

强制走从：

```
1. /*!mycat:db_type=slave*/ select * from travelrecord
2. /*#mycat:db_type=slave*/ select * from travelrecord
```

强制走写：

```
1. /*#mycat:db_type=master*/ select * from travelrecord
2. /*!mycat:db_type=master*/ select * from travelrecord
```

mysql主从备份是异步的复制，所以主从库存有一定的差距，在从库上进行的查询操作需要考虑到这些数据的差异，一般只有更新不频繁的数据或对实时性要求不高的数据可以通过从库查询，实时性要求高的数据仍然需要从主数据库获得。

为了减少主从复制的时延，也建议采用 MySQL 5.6+的版本，用 GTID 同步复制方式减少复制的时延。

对于某些表，要求不能有复制时延，则可以考虑这些表放到Gluster集群里，消除同步复制的时延问题。

MyCat1.4开始支持 MySQL 主从复制状态绑定的读写分离机制，让读更加安全可靠，配置如下：

- MyCAT 心跳检查语句配置为 show slave status；
- dataHost 上定义两个新属性: switchType="2" 与slaveThreshold="100"，此时意味着开启 MySQL 主从复制状态绑定的读写分离与切换机制。

Mycat 心跳机制通过读取"show slave status"中的 "Seconds_Behind_Master", "Slave_IO_Running", "Slave_SQL_Running" 三个字段来确定当前主从同步的状态，以及 Seconds_Behind_Master 主从复制时延，当 Seconds_Behind_Master>slaveThreshold 时，读写分离筛选器会过滤掉此 Slave 机器，防止读到很久之前的旧数据，而当主节点宕机后，切换逻辑会检查 Slave 上的 Seconds_Behind_Master 是否为 0，为 0 时则表示主从同步，可以安全切换，否则不会切换。

```
1. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="0" writeType="0" dbType="m
2.   <heartbeat>show slave status </heartbeat>
3.   <!-- can have multi write hosts -->
4.   <writeHost host="hostM1" url="localhost:3306" user="root" password="123456">
5.   </writeHost>
6.   <writeHost host="hostS1" url="localhost:3316" user="root" password="123456" />
7. </dataHost>
```

Mycat1.4.1 开始支持 galaxy cluster 集群的配置，提高心跳可用。

1.4.1开始支持MySQL 多主集群模式，让读更加安全可靠，配置如下： MyCAT 心跳检查语句配置为 show status like 'wsrep%'， dataHost 上定义两个新属性:switchType="3"。

此时意味着开启 MySQL 集群复制状态绑定的读写分离与切换机制， Mycat 心跳机制通过检测集群复制时延时，如果延时过大或者集群出现节点问题不会负载改节点。

```
1. <dataHost name="localhost1" maxCon="1000" minCon="10" balance="0" writeType="0" dbType="m
2.   <heartbeat> show status like 'wsrep%'</heartbeat>
3.   <writeHost host="hostM1" url="localhost:3306" user="root" password="123456"> </writeH
4.   <writeHost host="hostS1" url="localhost:3316" user="root" password="123456" ></writeH
5. </dataHost>
```

Mycat 里的数据库事务

Mycat 目前没有出来跨分片的事务强一致性支持，目前单库内部可以保证事务的完整性，如果跨库事务，在执行的时候任何分片出错，可以保证所有分片回滚，但是一旦应用发起 commit 指令，无法保证所有分片都成功，考虑到某个分片挂的可能性不大所以称为弱 xa。

MyCat扩容容、数据迁移

利用MyCat自带脚本dataMigrate.sh进行离线扩容容

1.复制schema.xml、rule.xml并重命名为newSchema.xml、newRule.xml，修改newSchema.xml和newRule.xml配置文件为扩容容后的mycat配置参数(表的节点数、 数据源、 路由规则)；

- 修改 conf 目录下的 migrateTables.properties 配置文件，告诉工具哪些表需要进行扩容或缩容，没有出现在此配置文件的 schema 表不会进行数据迁移；
- 如果可以确保扩容过程中不会有写操作，也可以不停止mycat服务；
- 执行 bin/dataMigrate.sh 脚本。

使用mysqldump进行数据迁移，和mysql没区别，直接dump进文本文件。

Influxdb

时序数据是基于时间的一系列的数据。

时序数据库就是存放时序数据的数据库，并且需要支持时序数据的快速写入、持久化、多纬度的聚合查询等基本功能。

下面介绍下时序数据库的一些基本概念(不同的时序数据库称呼略有不同)。

- metric: 度量，相当于关系型数据库中的table。
- data point: 数据点，相当于关系型数据库中的row。
- timestamp: 时间戳，代表数据点产生的时间。
- field: 度量下的不同字段。比如位置这个度量具有经度和纬度两个field。一般情况下存放的是会随着时间戳的变化而变化的数据。
- tag: 标签，或者附加信息。一般存放的是并不随着时间戳变化的属性信息。timestamp加上所有的tags可以认为是table的primary key。

如下图，度量为Wind，每一个数据点都具有一个timestamp，两个field: direction和speed，两个tag: sensor、city。它的第一行和第三行，存放的都是sensor号码为95D8-7913的设备，属性城市是上海。随着时间的变化，风向和风速都发生了改变，风向从23.4变成23.2;而风速从3.4变成了3.3。

timestamp	direction	speed	sensor	city
1467627245000	23.4	3.4	95D8-7913	上海
1467627245000	145.1	1.1	F3CC-20F3	北京
1467627247000	23.2	3.3	95D8-7913	上海
1467627247000	145.0	1.2	F3CC-20F3	北京
1467627255000	23.3	3.3	95D8-7913	上海
1467627255000	145.2	1.2	F3CC-20F3	北京

p3-时序数据库基本概念图

传统数据库存储采用的都是B tree，这是由于其在查询和顺序插入时有利于减少寻道次数的组织形式。时序数据库采用的LSM Tree，比如Hbase, Cassandra等nosql中。

InfluxDB有三大特性：

- Time Series (时间序列)：你可以使用与时间有关的相关函数（如最大，最小，求和等）
- Metrics (度量)：你可以实时对大量数据进行计算
- Events (事件)：它支持任意的事件数据

InfluxDB 支持两种api方式

HTTP API, InfluxDB 增删查都是用http api来完成.

Protobuf API, 还没完全实现。

如何使用 http api 进行操作?

比如对于foo_production这个数据库, 插入一系列数据, 可以发现POST 请求到 /db/foo_production/series?u=some_user&p=some_password, 数据放到body里。

数据看起来是这样的:

下面的"name": "events", 其中"events"就是一个series, 类似关系型数据库的表table

```
1. [
2.   {
3.     "name": "events",
4.     "columns": ["state", "email", "type"],
5.     "points": [
6.       ["ny", "paul@influxdb.org", "follow"],
7.       ["ny", "todd@influxdb.org", "open"]
8.     ]
9.   },
10.  {
11.    "name": "errors",
12.    "columns": ["class", "file", "user", "severity"],
13.    "points": [
14.      ["DivideByZero", "example.py", "someguy@influxdb.org", "fatal"]
15.    ]
16.  }
17. ]
```

上面的数据里没有包含time 列, InfluxDB会自己加上, 不过也可以指定time.

InfluxDB 提供了类似sql的查询语言

```
1. select * from events where state == 'NY';
2. select * from log_lines where line =~ /error/i;
3. select * from events where customer_id == 23 and type == 'click';
4. select * from response_times where value > 500;
5. select * from events where email !~ /.*gmail.*/;
6. select * from nagios_checks where status != 0;
7. select * from events where (email =~ /.*gmail.* or email =~ /.*yahoo.*/) and state == 'ny'
8. delete from response_times where time > now() - 1h
```

Redis

Redis与Memcached区别

- 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。
- Redis支持数据的备份, 即master-slave模式的数据备份。
- 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。
- Redis只会缓存所有的 key的信息, 如果Redis发现内存的使用量超过了某一个阀值, 将触发swap的操作, Redis会计算出哪些key对应的value需要swap到磁盘。然后再将这些key对应的value持久化到磁盘中, 同时在内存中清除。(看Redis内存限制即回收key策略)

- Memcached是多线程，非阻塞IO复用的网络模型。Redis使用单线程的IO复用模型，主要实现了epoll、kqueue和select。
- memcached的简单限制就是键（key）和item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。Redis的key 512K，值最大为1G。

Redis逻辑集群介绍

Redis Cluster是一种服务器Sharding技术，3.0版本开始正式提供。Sharding采用slot(槽)的概念，一共分成16384个槽。对于每个进入Redis的键值对，对key使用hash算法散列，分配到这16384个slot中的某一个中。N个主各划分所有槽，每个主又有n个slave从节点。比如有3个机器，我们准备搭一个3主2从的架构，3个主依次划分0-5461, 5462-10922, 10923-16384，每个主下面又挂一个备。根据gossip无中心选举算法，当哪个主挂掉后，n个slave从节点。强烈建议要1:1的模式，条件允许的情况下，建议1:n的模式。

庆幸的是，java redis客户端驱动jedis，已支持Redis Sharding功能，即ShardedJedis以及结合缓存池的ShardedJedisPool。

Redis Cluster采用无中心结构，每个节点都保存数据和整个集群的状态

每个节点都和其他所有节点连接，这些连接保持活跃。节点的fail是通过集群中超过半数的节点检测失效时才生效。客户端与redis节点直连，不需要中间proxy层。客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可。redis-cluster把所有的物理节点映射到[0-16383]slot上，cluster 负责维护node<->slot<->value。

使用gossip协议传播信息以及发现新节点。3.0以前使用的raft数据一致性算法，现在把raft改造，符合选多主的算法gossip。

选举过程是集群中所有master参与，如果半数以上master节点与master节点通信超过，认为当前master节点挂掉。

node不作为client请求的代理，client根据node返回的错误信息重定向请求

预分好16384个桶，根据 $\text{CRC16}(\text{key}) \bmod 16384$ 的值，决定将一个key放到哪个桶中。每个Redis物理结点负责一部分桶的管理，当发生Redis节点的增减时，调整桶的分布即可。

为了保证服务的可用性，Redis Cluster采取的方案是的Master-Slave。每个Redis Node可以有一个或者多个Slave。当Master挂掉时，选举一个Slave形成新的Master。一个Redis Node包含一定量的桶，当这些桶对应的Master和Slave都挂掉时，这部分桶对应的数据不可用。

Redis Cluster使用异步复制，一个完整的写操作步骤：

1. client写数据到master
2. master告诉client "ok"
3. Redis Cluster支持在线增/减节点。

基于桶的数据分布方式大大降低了迁移成本，只需将数据桶从一个Redis Node迁移到另一个Redis Node即可完成迁移。

当桶从一个Node A向另一个Node B迁移时，Node A和Node B都会有这个桶，Node A上桶的状态设置为MIGRATING，Node B上桶的状态被设置为IMPORTING

当客户端请求时：所有在Node A上的请求都将由A来处理，所有不在A上的key都由Node B来处理。同时，Node A上将不会创建新的keymaster传播更新到slave。

当我们以任意连接方式为集群加入一个节点，集群中所有节点都会自动对新节点建立信任连接。也就是说，集群具备自动识别所有节点的功能，但是这仅发生在当系统管理强制为新节点与集群中任意节点建立信任连接的前提下。Redis客户端被允许向集群中的任意节点发送命令，其中包括从节点。被访问的节点将会分析命令中传来的

key，并自己通过判断hash slot确定数据存储于哪个节点。如果被请求节点拥有hash slot数据（这里指请求数据未被迁移过或者hash slot在数据迁移后被重新计算过），则会直接返回结果，否则将会返回一个MOVED错误。

MOVED 错误如下：

```
1. GET x  
2. -MOVED 3999 127.0.0.1:6381
```

这个错误包括请求的数据所处的 hash slot (3999) 和 数据目前所属的IP:端口。客户端需要通过访问返回的IP:端口获取数据。即使在客户端请求并等待数据返回的过程中，集群配置已被更改，目标节点依然会返回一个MOVED错误。所以虽然在集群中的节点使用节点ID来确定身份，但是map依然是靠hash slot和Redis节点的IP:端口来进行配对。客户端虽然不被要求但是应该尝试去记住hash slot 3999现在已被转移至127.0.0.1:6381。这样的话，当一个新的命令需要从hash slot 3999获取数据时就可以有更高的几率从hash slot获取到正确的目标节点。假设集群已经足够的稳定（不增删节点），那么所有的客户端将会拥有一份hash slots对应节点的数据，这可以使整个集群更高效，因为这样每个命令都会直接定向到正确的节点，不需要通过节点寻找节点或者重新计算hash slot对应的节点。

环境搭建：

选两台机器，每个机器启动三个redis进程，启动时指定各自的redis.conf。

```
1. cd /usr/software  
2. redis-server redis_cluster/7000/redis.conf  
3. redis-server redis_cluster/7001/redis.conf  
4. redis-server redis_cluster/7002/redis.conf  
5. redis-server redis_cluster/7003/redis.conf  
6. redis-server redis_cluster/7004/redis.conf  
7. redis-server redis_cluster/7005/redis.conf
```

每个redis.conf结构如下：

```
1. daemonize yes //redis后台运行  
2. pidfile /var/run/redis_7000.pid //pidfile文件对应7000,7001,7002  
3. port 7000 //端口7000,7002,7003  
4. cluster-enabled yes //开启集群 把注释#去掉  
5. cluster-config-file nodes_7000.conf //集群的配置 配置文件首次启动自动生成 7000,7001,7002  
6. bind 192.168.215.130 //这里要绑定机器的IP  
7. cluster-node-timeout 5000 //请求超时 设置5秒够了  
8. appendonly yes //aof日志开启 有需要就开启，它会每次写操作都记录一条日
```

redis的集群是逻辑集群，通过一系列命令组成一个集群环境，为此，redis专门准备了1个ruby写的创建集群脚本

```
1. ./redis-trib.rb create --replicas 1 192.168.215.129:7000 192.168.215.129:7001 192.16
```

--replicas 1 表示 自动为每一个master节点分配一个slave节点 上面有6个节点，程序会按照一定规则生成 3个 master（主）3个slave(从)

redis集群中数据分片是通过hash slot的方式实现的

链接服务器

```
1. [root@zk1 src]# ./redis-cli -c -p 7000 -h 192.168.215.129  
2. 192.168.215.129:7000>  
3. 192.168.215.129:7000> cluster info
```

```
4. cluster_state:ok
5. cluster_slots_assigned:16384
6. cluster_slots_ok:16384
7. cluster_slots_pfail:0
8. cluster_slots_fail:0
9. cluster_known_nodes:6
10. cluster_size:3
11. cluster_current_epoch:6
12. cluster_my_epoch:1
13. cluster_stats_messages_sent:1502
14. cluster_stats_messages_received:1502
```

set值

```
1. 192.168.215.129:7000> set name lbl
2. -> Redirected to slot [5798] located at 192.168.215.130:7003
3. 0K
4. 192.168.215.130:7003> get name
5. "lbl"
```

可见，重定向到了130节点7003端口。

原因是redis采用hash槽的方式分发key到不同节点，算法是 $\text{crc}(16) \% 16384$ 。详细描述后续会单独写文章描述。
而且你会发现，当一次重定向以后，这个客户端就连接到了130:7003这个节点。
当有哪个主宕掉后，自动的原来这个主的从会选成它的主。

新的Redis可以配置密码，所以jedis客户端要使用比较新的版本。这里redis版本是3.2.5， jedis版本是2.9.0

Redis 基础

1. Redis数据类型及操作：

字符串 (Strings)

列表 (List)

集合 (Set)

哈希 / 散列 (Hash)

有序集合 (Sorted set)

位图 (Bitmap) 和超重对数 (HyperLogLog)

Redis超重对数HyperLogLog 是用来做基数统计的算法。什么是基数？比如数据集 {1, 3, 5, 7, 5, 7, 8}，那么这个数据集的基数集为 {1, 3, 5, 7, 8}，基数(不重复元素)为5。

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 事务可以一次执行多个命令。

2. 多数据库特性

1) 每个数据库对外都是以一个从0开始的递增数字命名，不支持自定义的。

2) redis默认支持16个数据库，可以通过修改databases参数来修改这个默认值。

3) redis默认选择的是0号数据库。

4) SELECT 数字：可以切换数据库。

5) 多个数据库之间并不是完全隔离的。

6) flushall：清空redis实例下的所有数据。

7) flushdb：清空当前数据库中的所有数据。

Redis内存限制即回收key策略

如果一个Redis实例的内存使用率超过可用最大内存(used_memory > 可用最大内存)，那么操作系统开始进行内存与swap空间交换，把内存中旧的或不再使用的内容写入硬盘上（硬盘上的这块空间叫Swap分区），以便腾出新的物理内存给新页或活动页(page)使用。在硬盘上进行读写操作要比在内存上进行读写操作，时间上慢了近5个数量级。

若Redis启用了Redis快照功能，应该设置“maxmemory”值为系统可使用内存的45%，因为快照时需要一倍的内存来复制整个数据集，也就是说如果当前已使用45%，在快照期间会变成95%(45%+45%+5%)，其中5%是预留给其他的开销。如果没开启快照功能，maxmemory最高能设置为系统可用内存的95%。

若是在使用Redis期间没有开启rdb快照或aof持久化策略，那么缓存数据在Redis崩溃时就有丢失的危险。因为当Redis内存使用率超过可用内存的95%时，部分数据开始在内存与swap空间来回交换，这时就可能有丢失数据的危险。当开启并触发快照功能时，Redis会fork一个子进程把当前内存中的数据完全复制一份写入到硬盘上。因此若是当前使用内存超过可用内存的45%时触发快照功能，那么此时进行的内存交换会变得非常危险(可能会丢失数据)。

当内存使用达到设置的最大阀值时，需要选择一种key的回收策略：可在Redis.conf配置文件中修改“maxmemory-policy”属性值。或下面问题：

问题：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。

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。redis提供6种数据淘汰策略通过maxmemory-policy设置策略：

- 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，这种策略使得我们可以向Redis提示哪些key更适合被eviction。
- 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，如果我们的应用对于缓存key的访问概率相等，则可以使用这个策略。
- allkeys-lru：从数据集 (server.db[i].dict) 中挑选最近最少使用的数据淘汰，我们不太清楚我们应用的缓存访问分布状况，我们可以选择。
- allkeys-random：从数据集 (server.db[i].dict) 中任意选择数据淘汰
- no-eviction (驱逐)：禁止驱逐数据

值得一提的是将key设置过期时间实际上会消耗更多的内存，因此我们建议使用allkeys-lru策略从而更有效率的使用内存。

Redis应用已使用场景

1. redis中键的生存时间 (expire)

redis中可以使用expire命令设置一个键的生存时间，到时间后redis会自动删除它。

1. expire 设置生存时间 (单位/秒)
2. expire key seconds(秒)
- 3.
4. ttl 查看键的剩余生存时间
5. ttl key
- 6.
7. persist 取消生存时间
8. persist key

应用场景：

- 限时的优惠活动信息
- 网站数据缓存（对于一些需要定时更新的数据，例如：积分排行榜）
- 手机验证码

问题：Redis数据库可以对键值对设置过期时间，当键值对过期时，redis会通过一定的机制将过期键删除？redis的过期键删除策略有两种：定期删除和惰性删除。

1. 惰性删除 惰性删除是每次获取键值对时，都对获取的键进行过期检查，如果过期的话，就删除该键值对；如果没过期，就返回该键。

1) 惰性删除策略的好处是对cpu时间比较友好，每次只检查当前处理的键值对，不会对其他过期的键值对花费cpu时间。

2) 惰性删除策略的缺点就是对内存不友好，一个键已经过期，但是这个键还会留在数据库中，只要它不被访问，就不会被删除，一直占用这内存空间。如果这些过期键长期不被访问，将永远不会被删除，我们可以视作内存泄漏——无用的数据占据了大量内存，而服务器不去释放它。

当然，用户可以通过手动执行FLUSHDB来删除过期键。

2. 定期删除 定期删除是每隔一段时间，程序对数据库检查一次，删除数据库里的过期键。至于每次检查多少数据库，删除多少过期键，由算法决定。

定期删除策略的关键点就是删除操作执行的时长和频率：

- 1) 如果删除操作太过频繁或者执行时间太长，就对cpu时间不是很友好，cpu时间过多的消耗在删除过期键上。
- 2) 如果删除操作执行太少或者执行时间太短，就不能及时删除过期键，导致内存浪费。

2. 发布与订阅

生产者：发布频道 "ch1"，发布消息。

消费者：监听频道"ch1"，消费消息。可取消订阅频道。

```
1. // 订阅频道数据
2. public static void testSubscribe() {
3.     //连接Redis数据库
4.     Jedis jedis = new Jedis("192.168.33.130", 6379);
5.     JedisPubSub jedisPubSub = new JedisPubSub() {
6.
7.         // 当向监听的频道发送数据时，这个方法会被触发
8.         @Override
9.         public void onMessage(String channel, String message) {
10.             System.out.println("收到消息" + message);
11.             //当收到 "unsubscribe" 消息时，调用取消订阅方法
12.             if ("unsubscribe".equals(message)) {
13.                 this.unsubscribe();
14.             }
15.         }
16.
17.         // 当取消订阅指定频道的时候，这个方法会被触发
18.         @Override
19.         public void onUnsubscribe(String channel, int subscribedChannels) {
20.             System.out.println("取消订阅频道" + channel);
21.         }
22.
23.     };
24.     // 订阅之后，当前进程一致处于监听状态，当被取消订阅之后，当前进程会结束
25.     jedis.subscribe(jedisPubSub, "ch1");
26.
27. }
```

```

28. // 发布频道数据
29. public static void testPubSub() throws Exception {
30.     //连接Redis数据库
31.     Jedis jedis = new Jedis("192.168.33.130", 6379);
32.     //发布频道 "ch1" 和消息 "hello redis"
33.     jedis.publish("ch1", "hello redis");
34.     //关闭连接
35.     jedis.close();
36. }
37.

```

3. 限制网站访客访问频率

比如想知道什么时候封锁一个IP地址。使用k-v的v的原子递增保持计数。

比如设置1分钟内，只能访问10次。

```

1. public boolean testLogin(String ip) {
2.     String value = jedis.get(ip);
3.     if (value == null) {
4.         //初始化时设置IP访问次数为1
5.         jedis.set(ip, "1");
6.         //设置IP的生存时间为60秒，60秒内IP的访问次数由程序控制
7.         jedis.expire(ip, 60);
8.     } else {
9.         int parseInt = Integer.parseInt(value);
10.        //如果60秒内IP的访问次数超过10，返回false，实现了超过10次禁止的功能
11.        if (parseInt > 10) {
12.            return false;
13.        } else {
14.            //如果没有10次，可以自增
15.            jedis.incr(ip);
16.        }
17.    }
18.    return true;
19. }
20. @Test
21. public void test3() throws Exception {
22.     // 模拟用户的频繁请求
23.     for (int i = 0; i < 20; i++) {
24.         boolean result = testLogin("192.168.1.100");
25.         if (result) {
26.             System.out.println("正常访问");
27.         } else {
28.             System.err.println("访问受限");
29.         }
30.     }
31. }
32.

```

4. 各种计数

如商品维度计数（喜欢数，评论数，鉴定数，浏览数,etc），用户维度计数（动态数、关注数、粉丝数、喜欢商品数、发帖数等）

采用Redis 的类型: Hash

为product定义个key product:，为每种数值定义hashkey, 譬如喜欢数xihuan

```

1. 127.0.0.1:6379> HSET product:1231233 like 5
2. (integer) 1
3. 127.0.0.1:6379> HINCRBY product:1231233 like 1      喜欢数+1
4. 128.(integer) 6

```

```
5. 127.0.0.1:6379> HGETALL product:1231233
6. 1) "LIKE"
7. 2) "6"
8. 127.0.0.1:6379>
```

获取该商品的喜欢数

- 5. 用作缓存代替Memecached
- 6. 利用List数据结构实现分布式的消息队列
- 7. 使用sorted set实现复杂的场景
- 8. 排行榜及相关问题。

Redis优化

1. 通过设置maxmemory的值为45%或95%(取决于持久化策略)和回收策略“maxmemory-policy”为“volatile-ttl”或“allkeys-lru”(取决于过期设置), 可以比较准确的限制Redis最大内存使用率。
2. Redis是个单线程模型, 客户端过来的命令是按照顺序执行的, 所以想要一次添加多条数据的时候可以使用管道, 或者使用一次可以添加多条数据的命令。
redis的pipeline(管道)功能在命令行中没有, 但是redis是支持管道的, 在Java的客户端(jedis)中是可以使用的。在插入更多数据的时候, 管道的优势更加明显: 测试10万条数据的时候, 不使用管道要40秒, 实用管道378毫秒。

```
1. // 测试不使用管道
2. public static void testInsert() {
3.     long currentTimeMillis = System.currentTimeMillis();
4.     Jedis jedis = new Jedis("192.168.33.130", 6379);
5.     for (int i = 0; i < 1000; i++) {
6.         jedis.set("test" + i, "test" + i);
7.     }
8.     long endTimeMillis = System.currentTimeMillis();
9.     System.out.println(endTimeMillis - currentTimeMillis);
10. }
11. // 测试管道
12. public static void testPip() {
13.     long currentTimeMillis = System.currentTimeMillis();
14.     Jedis jedis = new Jedis("192.168.33.130", 6379);
15.     Pipeline pipelined = jedis.pipelined();
16.     for (int i = 0; i < 1000; i++) {
17.         pipelined.set("bb" + i, i + "bb");
18.     }
19.     pipelined.sync();
20.     long endTimeMillis = System.currentTimeMillis();
21.     System.out.println(endTimeMillis - currentTimeMillis);
22. }
```

Redis持久化、数据同步和恢复

持久化

Redis支持两种方式的持久化, 可以单独使用或者结合起来使用。

第一种: RDB方式 (redis默认的持久化方式)

第二种: AOF方式

Redis启动时默认会在启动的目录生成RDB持久化文件dump.rdb, 服务端下次启动时, 如果还存在这个dump.rdb文件, 则上次的数据还存在, 如果不是, 则不存在。这个可以在配置文件redis.conf中配置。

首先，我们应该明确持久化的数据有什么用，答案是用于重启后的数据恢复。Redis是一个内存数据库，无论是RDB还是AOF，都只是其保证数据恢复的措施。所以Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。

Redis的数据回写机制分同步和异步两种：

- 同步回写save是由主进程进行快照操作，会阻塞其它请求。在数据大的情况下会导致系统假死很长时间，所以一般不是推荐的。
- 异步回写bgsave是由redis执行fork函数复制出一个子进程来进行快照回写磁盘，回写结束后新进程自行关闭。这样做不需要阻塞主进程，系统不会假死，一般默认会采用这个方法。

RDB方式 的持久化是通过Snapshot快照存储，是默认的持久化方式。按照一定的策略周期性的将数据保存到磁盘。对应产生的数据文件为dump.rdb，通过配置文件中的save参数来定义快照的周期策略。

下面是默认的快照策略设置：

```
1. save 900 1    #当有一条Keys数据被改变时，900秒刷新到Disk一次  
2. save 300 10   #当有10条Keys数据被改变时，300秒刷新到Disk一次  
3. save 60 10000 #当有10000条Keys数据被改变时，60秒刷新到Disk一次
```

手动执行save或者bgsave命令让redis执行快照。

Redis的RDB文件总是周期性的新生成，并替换掉原RDB文件。所以在生成过程中，不影响原先文件。

RDB的优缺点

优点：由于存储的有数据快照文件，恢复数据很方便。后面有恢复实验。

缺点：一旦数据库出现问题，RDB文件中保存的数据并不是全新的。由于是周期性备份，从上次RDB文件生成到Redis停机这段时间的数据全部丢掉了。

在数据恢复方面：

RDB的启动时间会更短，原因有两个：

- RDB文件中每一条数据只有一条记录，不会像AOF日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。
- RDB文件的存储格式和Redis数据在内存中的编码格式是一致的，不需要再进行数据编码工作，所以在CPU消耗上要远小于AOF日志的加载。

AOF方式 的持久化比RDB方式有更好的持久化性。AOF方式的持久化是通过日志文件的方式，在使用AOF持久化方式时，Redis会将每一个收到的写命令都通过Write函数追加到文件中，类似于MySQL的binlog。当Redis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。默认情况下redis没有开启aof，可以通过参数appendonly参数开启。aof文件的保存位置和rdb文件的位置相同，默认的文件名是appendonly.aof。

AOF的完全持久化方式同时也带来了另一个问题，持久化文件会变得越来越大。比如我们调用INCR test命令100次，文件中就必须保存全部的100条命令，但其实99条都是多余的。因为要恢复数据库的状态其实文件中保存一条SET test 100就够了。为了压缩AOF的持久化文件，Redis提供了bgrewriteaof命令。

使用bgrewriteaof命令压缩AOF的持久化文件，Redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，将整个内存中的数据库内容用命令的方式重写了一个新的AOF文件，最后替换原来的文件，以此来实现控制AOF文件的增长。

关于这两种方式，官方建议：

通常，如果你要想提供很高的数据保障性，那么建议你同时使用两种持久化方式。如果你可以接受灾难带来的几分钟的数据丢失，那么你可以仅使用RDB。很多用户仅使用了AOF，但是我们建议，既然RDB可以时不时的给数据做个完整的快照，并且提供更快的重启，所以最好还是也使用RDB。在未来（长远计划）统一AOF和RDB成一种持久化模式。

在线上环境中，由于数据都设置有过期时间，采用AOF的方式会不太实用，过于频繁的写操作会使AOF文件增长到异常的庞大，大大超过了我们实际的数据量，这也会导致在进行数据恢复时耗用大量的时间。因此，可以在Slave上仅开启Snapshot来进行本地化，同时可以考虑将save中的频率调高一些或者调用一个计划任务来进行定期bgsave的快照存储，来尽可能的保障本地化数据的完整性。

主从同步、主从切换、数据恢复演示

持久化的数据的作用是用于重启后的数据恢复，我们有必要进行一次这样的灾难恢复模拟了。如果数据要做持久化又想保证稳定性，则建议留空一半的物理内存。因为在进行快照的时候，fork出来进行dump操作的子进程会占用与父进程一样的内存，真正的copy-on-write，对性能的影响和内存的耗用都是比较大的。目前，通常的设计思路是利用副本(Replication)机制来弥补aof、snapshot性能上的不足，达到了数据可持久化。即**Master上Snapshot和AOF都不做，来保证Master的读写性能，而Slave上则同时开启Snapshot和AOF来进行持久化，保证数据的安全性。**

首先，修改Master上的如下配置：

```
1. $ sudo vim /opt/redis/etc/redis_6379.conf
2.
3. #save 900 1 #禁用Snapshot
4. #save 300 10
5. #save 60 10000
6.
7. appendonly no #禁用AOF
```

接着，修改Slave上的如下配置：

```
1. $ sudo vim /opt/redis/etc/redis_6379.conf
2.
3. save 900 1 #启用Snapshot
4. save 300 10
5. save 60 10000
6.
7. appendonly yes #启用AOF
8. appendfilename appendonly.aof #AOF文件的名称
9. # appendfsync always
10. appendfsync everysec #每秒钟强制写入磁盘一次
11. # appendfsync no
12.
13. no-appendfsync-on-rewrite yes #在日志重写时，不进行命令追加操作
14. auto-aof-rewrite-percentage 100 #自动启动新的日志重写过程
15. auto-aof-rewrite-min-size 64mb #启动新的日志重写过程的最小值
```

分别启动Master与Slave

```
1. $ /etc/init.d/redis start
```

```
1. redis 127.0.0.1:6379> CONFIG GET save
2. 1) "save"
3. 2) ""
```

使用命令redis-trib.rb生成主从。

主从复制

Redis的RDB文件是Redis主从同步内部实现中的一环。

第一次Slave向Master: Slave向Master发出同步请求，Master先dump出rdb文件，然后将rdb文件全量传输给slave，然后Master把缓存的命令转发给Slave，初次同步完成。

第二次以及以后的同步: Master将变量的快照增量依次发送给各个Slave。但不管什么原因导致Slave和Master断开重连都会重复以上两个步骤的过程。

问题: 当使用Redis读写分离，从机作为读的情况，从机宕机或者和主机断开连接都需要重新连接主机，重新连接主机都会触发全量的主从复制，这时候主机会生成内存快照，主机依然可以对外提供服务，但是作为读的从机，就无法提供对外服务了，如果数据量大，恢复的时间会相当的长。

为了解决Redis主从Copy的问题，有如下两个解决方案：

1. 主动复制 所谓主动复制，就是业务层双写多个Redis，避开Redis自带的主从复制。但是自己干同步，就会产生一致性问题，为了保证主从一致，需要加入一系列的验证机制。而且这样的做法，会降低系统性能。
2. 修改源代码，支持增量同步 Redis写AOF文件，关闭Redis rewrite AOF文件功能，为了避免文件过大，可以自己实现文件分割功能。在业务低峰时期，生成内存快照，并记录快照时刻AOF所在的点。当从机重连的时候，从机发送同步命令给主机，主机收到命令后，把最新的快照文件发送给从机，从机从快照文件中恢复，并且获得了该快照对应的AOF点，从机将AOF点发送给主机，主机将AOF文件中该点之后的所有数据操作同步给从机，达到增量同步的效果。

然后通过以下脚本在Master中生成25万条数据：

```
1. dongguo@redis:/opt/redis/data/6379$ cat redis-cli-generate.temp.sh
2.
3. #!/bin/bash
4.
5. REDISCLI="redis-cli -a slavepass -n 1 SET"
6. ID=1
7.
8. while(( $ID<50001 ))
9. do
10. INSTANCE_NAME="i-2-$ID-VM"
11. UUID=`cat /proc/sys/kernel/random/uuid`
12. PRIVATE_IP_ADDRESS=10.`echo "$RANDOM % 255 + 1" | bc`.`echo "$RANDOM % 255 + 1" | bc`.
13. CREATED=`date "+%Y-%m-%d %H:%M:%S"`
14.
15. $REDISCLI vm_instance:$ID:instance_name "$INSTANCE_NAME"
16. $REDISCLI vm_instance:$ID:uuid "$UUID"
17. $REDISCLI vm_instance:$ID:private_ip_address "$PRIVATE_IP_ADDRESS"
18. $REDISCLI vm_instance:$ID:created "$CREATED"
19.
20. $REDISCLI vm_instance:$INSTANCE_NAME:id "$ID"
21.
22. ID=$(( $ID+1 ))
23. done
24. dongguo@redis:/opt/redis/data/6379$ ./redis-cli-generate.temp.sh
```

在数据的生成过程中，可以很清楚的看到Master上仅在第一次做Slave同步时创建了dump.rdb文件，之后就通过增量传输命令的方式给Slave了。

dump.rdb文件没有再增大。

```
1. dongguo@redis:/opt/redis/data/6379$ ls -lh
2. total 4.0K
3. -rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb
```

而Slave上则可以看到dump.rdb文件和AOF文件在不断的增大，并且AOF文件的增长速度明显大于dump.rdb文件。

```
1. dongguo@redis-slave:/opt/redis/data/6379$ ls -lh
2. total 24M
3. -rw-r--r-- 1 root root 15M Sep 27 12:06 appendonly.aof
4. -rw-r--r-- 1 root root 9.2M Sep 27 12:06 dump.rdb
```

等待数据插入完成以后，首先确认当前的数据量。

```
1. redis 127.0.0.1:6379> info
2.
3. redis_version:2.4.17
4. redis_git_sha1:00000000
5. redis_git_dirty:0
6. arch_bits:64
7. multiplexing_api:epoll
8. ...
9. expired_keys:0
10. evicted_keys:0
11. keyspace_hits:0
12. keyspace_misses:0
13. pubsub_channels:0
14. pubsub_patterns:0
15. latest_fork_usec:246
16. vm_enabled:0
17. role:master
18. slave0:10.6.1.144,6379,online
19. db1:keys=250000,expires=0
```

当前的数据量为25万条key，占用内存31.52M。

然后我们直接Kill掉Master的Redis进程，模拟灾难。

```
1. dongguo@redis:/opt/redis/data/6379$ sudo killall -9 redis-server
```

我们到Slave中查看状态：

```
1. redis 127.0.0.1:6379> info
2.
3. redis_version:2.4.17
4. redis_git_sha1:00000000
5. redis_git_dirty:0
6. arch_bits:64
7. multiplexing_api:epoll
8. gcc_version:4.4.5
9. ...
10. pubsub_channels:0
```

```
11. pubsub_patterns:0
12. latest_fork_usec:694
13. vm_enabled:0
14. role:slave
15. aof_current_size:17908619
16. aof_base_size:16787337
17. aof_pending_rewrite:0
18. aof_buffer_length:0
19. aof_pending_bio_fsync:0
20. master_host:10.6.1.143
21. master_port:6379
22. master_link_status:down
23. master_last_io_seconds_ago:-1
24. master_sync_in_progress:0
25. master_link_down_since_seconds:25
26. slave_priority:100
27. db1:keys=250000,expires=0
```

可以看到master_link_status的状态已经是down了，Master已经不可访问了。
而此时，Slave依然运行良好，并且保留有AOF与RDB文件。

下面我们将通过Slave上保存好的AOF与RDB文件来恢复Master上的数据。

首先，将Slave上的同步状态取消，避免主库在未完成数据恢复前就重启，进而直接覆盖掉从库上的数据，导致所有的数据丢失。

```
1. redis 127.0.0.1:6379> SLAVEOF NO ONE
2. OK
```

确认一下已经没有了master相关的配置信息：

```
1. redis 127.0.0.1:6379> INFO
2.
3. redis_version:2.4.17
4. redis_git_sha1:00000000
5. redis_git_dirty:0
6. arch_bits:64
7. multiplexing_api:epoll
8. ...
9. pubsub_patterns:0
10. latest_fork_usec:1119
11. vm_enabled:0
12. role:master
13. aof_current_size:17908619
14. aof_base_size:16787337
15. aof_pending_rewrite:0
16. aof_buffer_length:0
17. aof_pending_bio_fsync:0
18. db1:keys=250000,expires=0
```

在Slave上复制数据文件：

```
1. dongguo@redis-slave:/opt/redis/data/6379$ tar cvf /home/dongguo/data.tar *
2. appendonly.aof
3. dump.rdb
```

将data.tar上传到Master上，尝试恢复数据：

可以看到Master目录下有一个初始化Slave的数据文件，很小，将其删除。

```
1. dongguo@redis:/opt/redis/data/6379$ ls -l
2. total 4
3. -rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb
4. dongguo@redis:/opt/redis/data/6379$ sudo rm -f dump.rdb
```

然后解压缩数据文件：

```
1. dongguo@redis:/opt/redis/data/6379$ sudo tar xf /home/dongguo/data.tar
2. dongguo@redis:/opt/redis/data/6379$ ls -lh
3. total 29M
4. -rw-r--r-- 1 root root 18M Sep 27 01:22 appendonly.aof
5. -rw-r--r-- 1 root root 12M Sep 27 01:22 dump.rdb
```

启动Master上的Redis；

```
1. dongguo@redis:/opt/redis/data/6379$ sudo /etc/init.d/redis start
2. Starting Redis server...
```

查看数据是否恢复：

```
1. redis 127.0.0.1:6379> INFO
2.
3. redis_version:2.4.17
4. redis_git_sha1:00000000
5. redis_git_dirty:0
6. arch_bits:64
7. multiplexing_api:epoll
8. ...
9. expired_keys:0
10. evicted_keys:0
11. keyspace_hits:0
12. keyspace_misses:0
13. pubsub_channels:0
14. pubsub_patterns:0
15. latest_fork_usec:0
16. vm_enabled:0
17. role:master
18. db1:keys=250000,expires=0
```

可以看到25万条数据已经完整恢复到了Master上。

此时，可以放心的恢复Slave的同步设置了。

```
1. redis 127.0.0.1:6379> SLAVEOF 10.6.1.143 6379
2. OK
```

查看同步状态：

```
1. redis 127.0.0.1:6379> INFO
2.
3. redis_version:2.4.17
4. redis_git_sha1:00000000
5. redis_git_dirty:0
6. arch_bits:64
```

```
7. multiplexing_api:epoll
8. gcc_version:4.4.5
9. ...
10. aof_pending_bio_fsync:0
11. master_host:10.6.1.143
12. master_port:6379
13. master_link_status:up
14. master_last_io_seconds_ago:0
15. master_sync_in_progress:0
16. slave_priority:100
17. db1:keys=250000,expires=0
```

master_link_status显示为up，同步状态正常。

在此次的案例中，我们通过在Slave上启用了AOF与RDB来保障了数据，并恢复了Master。

数据恢复

在上面恢复的过程中，我们同时复制了AOF与RDB文件，那么到底是哪一个文件完成了数据的恢复呢？

实际上，当Redis服务器挂掉时，重启时将按照以下优先级恢复数据到内存：

1. 如果只配置AOF，重启时加载AOF文件恢复数据；
2. 如果同时配置了RDB和AOF，启动是只加载AOF文件恢复数据；
3. 如果只配置RDB，启动是将加载dump文件恢复数据。

也就是说，AOF的优先级要高于RDB，这也很好理解，因为AOF本身对数据的完整性保障要高于RDB。

在线上环境中，由于数据都设置有过期时间，采用AOF的方式会不太实用，过于频繁的写操作会使AOF文件增长到异常的庞大，大大超过了我们实际的数据量，这也会导致在进行数据恢复时耗用大量的时间。因此，可以在Slave上仅开启Snapshot来进行本地化，同时可以考虑将save中的频率调高一些或者调用一个计划任务来进行定期bgsave的快照存储，来尽可能的保障本地化数据的完整性。在这样的架构下，如果仅仅是Master挂掉，Slave完整，数据恢复可达到100%。如果Master与Slave同时挂掉的话，数据的恢复也可以达到一个可接受的程度。

扩容缩容

直接在线切换

读写分离

对于读操作，jedis直接连接从库，写操作，jedis直接连接主库。

或第三方proxy实现。

jedis坑

Zookeeper基本使用

Zookeeper数据模型：分层结构，树形结构的每个节点叫做Znode，每个Znode都有数据，也可以有子节点。数据变化时，版本号会递增。

每个ZNode数据节点，用于存储数据，数据节点分为：

持久节点(PERSISTENT): 一旦创建，除非主动调用删除操作，否则一直存储在zk上；

临时节点(EPHEMERAL): 与客户端的会话绑定，一旦客户端会话失败，这个客户端创建的所有临时节点都会被移除；

SEQUENTIAL Znode: 创建节点时，如果设置属性SEQUENTIAL，则会自动在节点名后面追加一个整型数字。

ZK中引入了watcher机制来实现了发布/订阅功能，能够让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者。

Watcher作用于Znode节点上，分为数据更新和子节点状态等通知。

Watcher中的事件：

NodeDataChanged事件 节点数据变化

NodeChildrenChanged事件 新增节点或者删除节点

AutoFailed事件 授权失败

Watcher设置后，一旦触发一次即会失效，如果需要一直监听，就需要再注册。

ACL（访问权限控制列表）：

CREATE: 创建子节点的权限

READ: 获取节点数据和子节点列表的权限

WRITE: 更新节点数据的权限

DELETE: 删除子节点的权限

ADMIN: 设置节点ACL的权限

zkclient使用：

```
1. //1.创建会话
2. public static void main(String[] args) {
3.     //zk集群的地址
4.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
5.
6.     /**
7.      * 创建会话
8.      * new SerializableSerializer() 创建序列化器接口，用来序列化和反序列化
9.      */
10.    ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
11.
12.    System.out.println("conneted ok!");
13.
14. }
15. //2.创建节点
16. public static void main(String[] args) {
17.     //zk集群的地址
18.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
19.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
20.
21.     System.out.println("conneted ok!");
22. }
```

```
23. User user = new User();
24. user.setId(1);
25. user.setName("testUser");
26.
27. /**
28. * "/testUserNode" : 节点的地址
29. * user: 数据的对象
30. * CreateMode.PERSISTENT: 创建的节点类型
31. */
32. String path = zkClient.create("/testUserNode", user, CreateMode.PERSISTENT);
33. //输出创建节点的路径
34. System.out.println("created path:"+path);
35. }
36. //3.获取节点中的数据
37. public static void main(String[] args) {
38.     //zk集群的地址
39.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
40.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
41.     System.out.println("conneted ok!");
42.
43.     Stat stat = new Stat();
44.     //获取 节点中的对象
45.     User user = zkClient.readData("/testUserNode",stat);
46.     System.out.println(user.getName());
47.     System.out.println(stat);
48. }
49. //4.判断节点是否存在
50. public static void main(String[] args) {
51.     //zk集群的地址
52.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
53.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
54.     System.out.println("conneted ok!");
55.
56.     boolean e = zkClient.exists("/testUserNode");
57.     //返回 true表示节点存在 , false表示不存在
58.     System.out.println(e);
59. }
60. //5.删除节点
61. public static void main(String[] args) {
62.     //zk集群的地址
63.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
64.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
65.     System.out.println("conneted ok!");
66.
67.     //删除单独一个节点, 返回true表示成功
68.     boolean e1 = zkClient.delete("/testUserNode");
69.     //删除含有子节点的节点
70.     boolean e2 = zkClient.deleteRecursive("/test");
71.
72.     //返回 true表示节点成功 , false表示删除失败
73.     System.out.println(e1);
74. }
75. //6.更新数据
76. public static void main(String[] args) {
77.     //zk集群的地址
78.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
79.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
80.     System.out.println("conneted ok!");
81.
82.     User user = new User();
83.     user.setId(2);
```

```

84.         user.setName("testUser2");
85.         /**
86.          * testUserNode 节点的路径
87.          * user 传入的数据对象
88.         */
89.         zkClient.writeData("/testUserNode", user);
90.     }
91. //7.订阅节点的信息改变 (创建节点, 删除节点, 添加子节点)
92. private static class ZKChildListener implements IZkChildListener{
93.     /**
94.      * handleChildChange: 用来处理服务器端发送过来的通知
95.      * parentPath: 对应的父节点的路径
96.      * currentChilds: 子节点的相对路径
97.      */
98.     public void handleChildChange(String parentPath, List<String> currentChilds) throws E
99.
100.        System.out.println(parentPath);
101.        System.out.println(currentChilds.toString());
102.
103.    }
104.
105. }
106.
107. public static void main(String[] args) throws InterruptedException {
108.     //zk集群的地址
109.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
110.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
111.     System.out.println("conneted ok!");
112.     /**
113.      * "/testUserNode" 监听的节点, 可以是现在存在的也可以是不存在的
114.      */
115.     zkClient.subscribeChildChanges("/testUserNode3", new ZKChildListener());
116.     Thread.sleep(Integer.MAX_VALUE);
117. }
118. //8.订阅节点的数据内容的变化
119. private static class ZKDataListener implements IZkDataListener{
120.     public void handleDataChange(String dataPath, Object data) throws Exception {
121.         System.out.println(dataPath+":"+data.toString());
122.     }
123.     public void handleDataDeleted(String dataPath) throws Exception {
124.         System.out.println(dataPath);
125.     }
126. }
127.
128. public static void main(String[] args) throws InterruptedException {
129.     //zk集群的地址
130.     String ZKServers = "192.168.30.164:2181,192.168.30.165:2181,192.168.30.166:2181";
131.     ZkClient zkClient = new ZkClient(ZKServers,10000,10000,new SerializableSerializer());
132.     System.out.println("conneted ok!");
133.
134.     zkClient.subscribeDataChanges("/testUserNode", new ZKDataListener());
135.     Thread.sleep(Integer.MAX_VALUE);
136. }
137. }
```

Zookeeper典型应用场景

1.利用Zookeeper实现配置管理:

利用发布/订阅实现，发送者将数据发布到zk的一个或者一系列节点上，订阅者进行数据订阅，当数据变化时，可以及时得到数据的变化通知。

2.master选举：

- 1)zk上创建节点/servers节点；
- 2)各个服务器在启动过程中，首先会到ZK节点下的servers节点下，去创建一个临时节点，并把自己的基本信息写入到这个临时节点中，这个过程叫服务注册；
- 3)紧接着这些服务器会尝试着去创建/master临时节点，谁能创建成功谁就是master节点，其他的两台机子就作为slave。
- 4)所有的服务器必须监听/master节点的删除事件（因为ZK的临时节点会话失效后就会立刻被删除），一旦master节点宕机，其他的节点就会立刻发现，然后重新选举。

3.Zookeeper的发布和订阅：

发布订阅模式可以看成一对多的关系：多个订阅者对象同时监听一个主题对象，这个主题对象在自身状态发生变化时，会通知所有的订阅者对象，使他们能够自动的更新自己的状态。

发布订阅模式在分布式系统的典型应用有，配置管理和服务发现。

4.ZooKeeper 实现分布式锁：

分布式锁：分布式锁指的是在分布式环境下，保护跨进程，跨主机，跨网络的共享资源，实现互斥访问，保证一致性。

- 1)客户端在/lockers/node节点下创建顺序节点，返回一个顺序节点node_k；
- 2)立刻获取所有/lockers/下所有节点，看k之前是否有节点，如果有，说明有其他客户端都拿到锁了，这时阻塞监听node_(k-1)节点删除事件，如果监听到，当前客户端就拿到了锁，其他客户端也在watch这个节点。

5.ZooKeeper 实现分布式队列

1)生产者通过在/queue节点下创建顺序节点来存放数据；

2)消费者读取queue下所有子节点列表children，按由小到大的排序，读取最小的那个子节点的数据，读取到数据后，删除该节点。

6.ZooKeeper 实现命名服务

可以理解成一个分布式的ID生成器

命名服务可以理解为提供名字的服务

Zookeeper的命名服务，有两个应用方向：

1.提供类似JNDI的功能：

利用zookeeper中的树形分层结构，可以把系统中的各种服务的名称，地址以及目录信息存放在zookeeper中，需要的时候去zookeeper中去读取

2.利用zookeeper中的顺序节点的特性，制作分布式的序列号生成器（ID生成器）。创建持久的顺序节点，返回节点名称，删除该节点。

Zookeeper使用遇到问题

Watch是一次性触发器，如果你得到了一个watch事件，而你希望在以后发生变更时继续得到通知，你应该再设置一个watch，当然ZKClient已经帮你实现了这个事情。因为watch是一次性触发器，而获得事件再发送一个新的设置watch的请求这一过程会有延时，所以你无法确保你看到了所有发生在ZooKeeper上的一个节点上的事件。所以请处理好在这个时间窗口中可能会发生多次znode变更的这种情况。当你从一个服务器上断开时（比如服务器出故障

了），在再次连接上之前，你将无法获得任何watch。请使用这些会话事件来进入安全模式：在disconnected状态下你将不会收到事件，所以你的程序在此期间应该谨慎行事。

etcd基本使用

etcd作为HTTP+JSON的API，用curl命令可以轻松使用，提供可选SSL客户端认证机制。

etcdctl包含了etcd所有的操作，基本等同于ZooKeeper的功能。

```
1. #设置一个键的值
2. etcdctl set /foo/bar "hello world"
3. hello world
4. #设置一个带time to live的键
5. etcdctl set /foo/bar "hello world" --ttl 60
6. hello world
7. #当值为"hello world"时，替换为"Goodbye world"。
8. etcdctl set /foo/bar "Goodbye world" --swap-with-value "hello world"
9. etcdctl set /foo/bar "Goodbye world" --swap-with-value "hello world"
10. Goodbye world
11. etcdctl set /foo/bar "Goodbye world" --swap-with-value "hello world"
12. Error: 101: Compare failed ([hello world != Goodbye world]) [7]
13. #仅当不存在时创建
14. etcdctl mk /foo/new_bar "Hello world"
15. Hello world
16. etcdctl mk /foo/new_bar "Hello world"
17. Error: 105: Key already exists (/foo/new_bar) [8]
18. #仅当存在时更新键
19. etcdctl update /foo/bar "hello etcd"
20. hello etcd
21. #创建一个directory node
22. etcdctl mkdir /foo/dir
23. #创建一个directory node，或者将一个file node设置成directory node
24. etcdctl setDir /foo/dir
25. #删除file node
26. etcdctl rm /foo/bar
27. #删除directory node
28. etcdctl rmdir /foo/dir
29. etcdctl rm /foo/dir --dir
30. #递归删除
31. etcdctl rm /foo/dir --recursive
32. #当值为"Hello world"时删除
33. etcdctl rm /foo/bar --with-value "Hello world"
34. #获得某个键的值
35. etcdctl get /foo/bar
36. hello, etcd
37. #获得某个键在集群内的一致性值
38. etcdctl get /foo/bar --consistent
39. hello, etcd
40. #获得一些扩展的元信息
41. etcdctl -o extended get /foo/bar
42. Key: /foo/bar
43. Created-Index: 14
44. Modified-Index: 14
45. TTL: 0
46. Etcd-Index: 14
47. Raft-Index: 5013
48. Raft-Term: 0
49.
50. hello, etcd
51. #其中，索引是一个对于etcd上任何改变中唯一、单调递增的整数。这个特殊的索引反映了etcd在某个key被创建后的时
```

```
52. #列出目录的内容,-p则对directory node以/结尾
53. etcdctl ls
54. /foo
55. etcdctl ls /foo
56. /foo/bar
57. /foo/new_bar
58. /foo/dir
59. etcdctl ls / --recursive
60. /foo
61. /foo/bar
62. /foo/new_bar
63. /foo/dir
64.
65. #设置监视(watch),此时该命令会一直等待并输出下一次变化。
66. etcdctl watch /foo/bar
67. hello, etcd!
68. #while another terminal
69. etcdctl update /foo/bar "hello, etcd!"
70. hello, etcd!
71. #持续监视更新
72. etcdctl watch /foo/bar --forever
73. .....
74. #使用Ctrl+c结束
75. #当发生变化时执行一个应用
76. etcdctl exec-watch /foo/bar -- sh -c "echo hi"
77. hi
78. hi
79. .....
80. #监视目录下所有节点的改变
81. etcdctl exec-watch --recursive /foo -- sh -c "echo hi"
```

通过curl维护etcd

```
1. #获得版本
2. curl -L http://127.0.0.1:4001/version
3. etcd 0.4.6
4. 查看集群信息
5. # 查看leader
6. curl -L http://127.0.0.1:4001/v2/leader
7. http://127.0.0.1:7001
8. # 查看peers
9. curl -L http://127.0.0.1:4001/v2/machines
10. http://127.0.0.1:4001, http://127.0.0.1:4002, http://127.0.0.1:4003
11. # 获得一致性get
12. etcdctl set /foo/bar "Hi, etcd cluster"
13. Hi, etcd cluster
14. etcdctl get /foo/bar --consistent
15. Hi, etcd cluster
16. # 干掉etcd1(http://127.0.0.1:4001)服务进程后, 检查leader。已经成功切换到etcd3
17. curl -L http://127.0.0.1:4002/v2/leader
18. http://127.0.0.1:7003
19. #set操作, URL为http://127.0.0.1:4001/v2/keys
20. curl -L http://127.0.0.1:4001/v2/keys/foo/bar -XPUT -d value="hi, etcd"
21. {"action":"set","node":{"key":"/foo/bar","value":"hi, etcd","modifiedIndex":27,"createdIndex":26}
22. #get操作
23. curl -L http://127.0.0.1:4001/v2/keys/foo/bar
24. {"action":"get","node":{"key":"/foo/bar","value":"hi, etcd","modifiedIndex":27,"createdIndex":26}
25. #rm操作
26. curl -L http://127.0.0.1:4001/v2/keys/foo/bar -XDELETE
27. {"action":"delete","node":{"key":"/foo/bar","modifiedIndex":28,"createdIndex":27}, "prevNodeKey": null}
28. #获得版本
```

```
29. curl -L http://127.0.0.1:4001/version
30. etcd 0.4.6
31. #set操作, URL为http://127.0.0.1:4001/v2/keys
32. curl -L http://127.0.0.1:4001/v2/keys/foo/bar -XPUT -d value="hi, etcd"
33. {"action":"set","node":{"key":"/foo/bar","value":"hi, etcd","modifiedIndex":27,"createdIndex":27}
34. #get操作
35. curl -L http://127.0.0.1:4001/v2/keys/foo/bar
36. {"action":"get","node":{"key":"/foo/bar","value":"hi, etcd","modifiedIndex":27,"createdIndex":27}
37. #rm操作
38. curl -L http://127.0.0.1:4001/v2/keys/foo/bar -XDELETE
39. {"action":"delete","node":{"key":"/foo/bar","modifiedIndex":28,"createdIndex":27}, "prevNo
```

etcd使用场景

- 1.服务发现：借助一个强一致性/高可用的服务存储目录，实现服务动态添加。
- 2.消息发布与订阅 数据提供者发布消息，消费者订阅
- 3.软负载均衡
- 4.分布式通知与协调 使用watch具体目录。
- 5.分布式锁
- 6.分布式队列

etcd使用遇到问题

无

Zookeeper和etcd和consul对比

zookeeper与etcd对比：

zk复杂，使用Paxos强一致性算法，zk使用也比较复杂，需要安装客户端。

java编写

etcd简单，使用go编写，部署简单，作为HTTP+JSON的API，用curl命令可以轻松使用，提供可选SSL客户端认证机制。

Zookeeper提供了临时节点，sequence，和变更通知。利用Zookeeper的这3个特性实现了按照sequence的顺序依次获取锁和成为主。

etcd没有临时节点的概念，但是通过租约的方式提供了类似的功能。etcd没有sequence的概念，但是提供了全局递增的序列号revision，通过判断每个key的revision，也可以实现类似的sequence功能。提供多键条件事务（类似etcdv2的Compare-and-Swap）。虽然提供的机制不同与Zookeeper，但是实现的锁方式和选主方式与zookeeper非常类似，也是按照key建立时间依次获得锁和成为主。

Paxos/ZAB/raft协议对比

zab协议

zookeeper是基于paxos的简化版zab，ZAB即Zookeeper原子消息广播协议，选举过程和数据写入过程都需要依赖此协议。写只在一个节点去写，一个数据的变更，由两个提议组成，一个提议是对数据修改的本身，另一个提议是数据事务的提交。

服务器的三种角色：

Leader: 事务请求的唯一调度和处理者，集群内各服务器的调度者

Follower: 处理客户端非事务请求；转发事务请求给Leader；参与事务请求Proposal的投票；参与Leader选举投票。

Observer: 处理客户端非事务请求；转发事务请求给Leader；不存于任何形式（事务请求和选举Leader）投票。

服务器的状态：

LOOKING: 寻找Leader状态，当处于当前状态时，表示没有Leader，需要进入选举流程；

FOLLOWING: 当前服务器已经是Follower。

OBSERVING: 当前服务器时Observer角色。

LEADING: 当前服务器时Leader。

ZAB协议的三个阶段：

1.发现，即选举Leader过程；

2.同步，选举出新的Leader后，Follower或者Observer从Leader同步最新的数据；

3.广播，同步完成后，就可以接收客户端新的事务请求，并进行消息广播，实现数据在集群节点的副本存储。

ZAB具体流程：

为避免两个节点之间的TCP连接，zk按照myid数值方向建立连接，比如id为1的向id为2的发起tcp连接。并且数据同步端口2888，投票端口3888，分开的。

集群启动前，每个都有一个myid文件，myid中数字越大的就会被选举成为Leader，当集群中已经有Leader时，新加入的节点不会影响原来的集群。通过事务id(zxid)的大小来表示数据的新旧，越大代表数据越新。

1.初次启动，三个zk节点，对应的myid为1,2,3。

1)启动myid为1的节点，此时zxid为0，此时没法选举出主节点。

2)启动myid为2的节点，它的zxid也为0，此时2这个节点成为主节点

3)启动myid为3的节点，因为已经有主节点，则3加入集群，2还是Leader.

当Leader突然宕机后，其他节点的状态变更为LOOKING。

1)每个server发出一个投自己的票的投票。生成投票信息(myid,zxid)，server1为(1,123)，server3为(2,122)，server1发给server3，server3发给server1。

2)server3收到server1，因为server1的123比122大，所以，server3修改自己的投票为(1,123)，然后发给server1.server1收到server3的投票，因为123大于122，因此不改变自己的投票。

3)各个服务器开始统计自己的票，server3统计，自己收到的投票(包含自己投的)中，(1,123)是两票，server1统计，自己受到投票(包含自己投的)中，(1,123)只有两票。因此，server3选出的leader是1，而自己是3，因此自己进入following状态，即follower角色。server1，选出的leader是1，自己就是1，因此自己进入LEADING状态，即自己是leader角色。

4)当leader完成选举后，follower需要与新的leader同步数据，同步前准备：

Leader端，告诉其他follower当前最新数据是什么即zxid是什么。Leader构建一个NEWLEADER的包，包括当前最大的zxid，发送给所有的follower或者Observer。然后，Leader给每个follower创建一个线程LearnerHandler来负责处理每个follower的数据同步请求，同时主线程开始阻塞，只有超过一半的follower同步完成，同步过程才完成，leader才能成为真正的leader。

follower端，选举完成后，尝试与leader建立同步连接，如果一段时间没有连接上就报超时，重新回到选举状态。

集群选举完成，并且完成数据同步后，即可开始对外服务，接收读写请求：

Leader是唯一的可以对事务请求进行写操作的节点，客户端不仅可以连接Leader，还可以连接Follower，但是Follower节点收到写请求后，要把写的事务请求转发给Leader节点，Follower节点可以处理客户端的读请求。

1)当接收到客户端新的事务请求后，会生成对应的事务proposal，并根据zxid的顺序向所有的follower发送提案(proposal)；

- 2)当follower收到leader的事务proposal时，根据接收的先后顺序处理这些proposal，即如果先后收到1,2,3条，则如果处理完了第3条，则代表1,2两条一定已经处理成功；
- 3)当leader收到过半的follower针对某个事务proposal的ack后，则发送事务提交，重新发起一个commit的proposal；
- 4)follower收到commit的proposal后，记录事务提交，并把数据更新到内存数据库。

raft协议

etcd 集群的工作原理基于 raft 共识算法。raft 共识算法的优点在于可以在高效的解决分布式系统中各个节点日志内容一致性问题的同时，也使得集群具备一定的容错能力。即使集群中出现部分节点故障、网络故障等问题，仍可保证其余大多数节点正确的步进。甚至当更多的节点（一般来说超过集群节点总数的一半）出现故障而导致集群不可用时，依然可以保证节点中的数据不会出现错误的结果。

raft 集群中的每个节点都可以根据集群运行的情况在三种状态间切换：follower(选民，完全被动), candidate(候选人，类似Proposer律师，可以被选为一个新的领导人)与 leader。leader 向 follower 同步日志，follower 只从 leader 处获取日志。

在 raft 集群中，所有日志都必须首先提交至 leader 节点。leader 在每个 heartbeat 向 follower 同步日志，follower 在收到日志之后向 leader 反馈结果，leader 在确认日志内容正确之后将此条目提交并存储于本地磁盘。

- 1)首先有一条 uncommitted 的日志条目提交至 leader 节点。
- 2)在下一个 heartbeat，leader 将此条目复制给所有的 follower。
- 3)当大多数节点记录此条目之后，leader 节点认定此条目有效，将此条目设定为已提交并存储于本地磁盘。
- 4)在下一个 heartbeat，leader 通知所有 follower 提交这一日志条目并存储于各自的磁盘内。

1. 集群建立与状态机（集群初始状态）

在节点初始启动时，节点的 raft 状态机将处于 follower 状态并被设定一个 election timeout，如果在这一时间周期内没有收到来自 leader 的 heartbeat，节点将发起选举：节点在将自己的状态切换为 candidate 之后，向集群中其它 follower 节点发送请求，询问其是否选举自己成为 leader。当收到来自集群中过半数节点的接受投票后，节点即成为 leader，开始接收保存 client 的数据并向其它的 follower 节点同步日志。leader 节点依靠定时向 follower 发送 heartbeat 来保持其地位。任何时候如果其它 follower 在 election timeout 期间都没有收到来自 leader 的 heartbeat，同样会将自己的状态切换为 candidate 并发起选举。每成功选举一次，新 leader 的步进数都会比之前 leader 的步进数大1。

下面为一个 etcd 集群选举过程的简单描述：

- 1)初始状态下集群中的所有节点都处于 follower 状态。
- 2)某一时刻，其中的一个 follower 由于没有收到 leader 的 heartbeat 率先发生 election timeout 进而发起选举。
- 3)只要集群中超过半数的节点接受投票，candidate 节点将成为即切换 leader 状态。
- 4)成为 leader 节点之后，leader 将定时向 follower 节点同步日志并发送 heartbeat。

2. 节点异常

节点的异常大致可以分为四种类型：leader 不可用；follower 不可用；多个 candidate 或多个 leader；新节点加入集群。

2.1 leader 不可用

- 1)一般情况下，leader 节点定时发送 heartbeat 到 follower 节点。
- 2)由于某些异常导致 leader 不再发送 heartbeat，或 follower 无法收到 heartbeat。
- 3)当某一 follower 发生 election timeout 时，其状态变更为 candidate，并向其他 follower 发起投票。

- 4)当超过半数的 follower 接受投票后，这一节点将成为新的 leader，leader 的步进数加1并开始向 follower 同步日志。
- 5)当一段时间之后，如果之前的 leader 再次加入集群，则两个 leader 比较彼此的步进数，步进数低的 leader 将切换自己的状态为 follower。
- 6)较早前 leader 中不一致的日志将被清除，并与现有 leader 中的日志保持一致。

2.2 follower 节点不可用

1)集群中的某个 follower 节点发生异常，不再同步日志以及接收 heartbeat。

2)经过一段时间之后，原来的 follower 节点重新加入集群。

3)这一节点的日志将从当时的 leader 处同步。

2.3 多个 candidate 或多个 leader

在集群中出现多个 candidate 或多个 leader 通常是由于数据传输不畅造成的。出现多个 leader 的情况相对少见，但多个 candidate 比较容易出现在集群节点启动初期尚未选出 leader 的“混沌”时期。

1)初始状态下集群中的所有节点都处于 follower 状态。

2)两个节点同时成为 candidate 发起选举。

3)两个 candidate 都只得到了少部分 follower 的接受投票。

4)candidate 继续向其他的 follower 询问。

5)由于一些 follower 已经投过票了，所以均返回拒绝接受。

6)candidate 也可能向一个 candidate 询问投票。

7)在步进数相同的情况下，candidate 将拒绝接受另一个 candidate 的请求。

8)由于第一次未选出 leader，candidate 将随机选择一个等待间隔（150ms ~ 300ms）再次发起投票。

9)如果得到集群中半数以上的 follower 的接受，这一 candidate 将成为 leader。

10)稍后另一个 candidate 也将再次发起投票。

11)由于集群中已经选出 leader，candidate 将收到拒绝接受的投票。

12)在被多数节点拒绝之后，并已知集群中已存在 leader 后，这一 candidate 节点将终止投票请求、切换为 follower，从 leader 节点同步日志。

一个 etcd 节点的核心由三部分组成：

Raft: raft 状态机是对 raft 共识算法的实现

WAL: raft 日志存储

Storage: 数据的存储与索引

WAL (Write-ahead logging)，是用于向系统提供原子性和持久性的一系列技术。

分布式文件系统

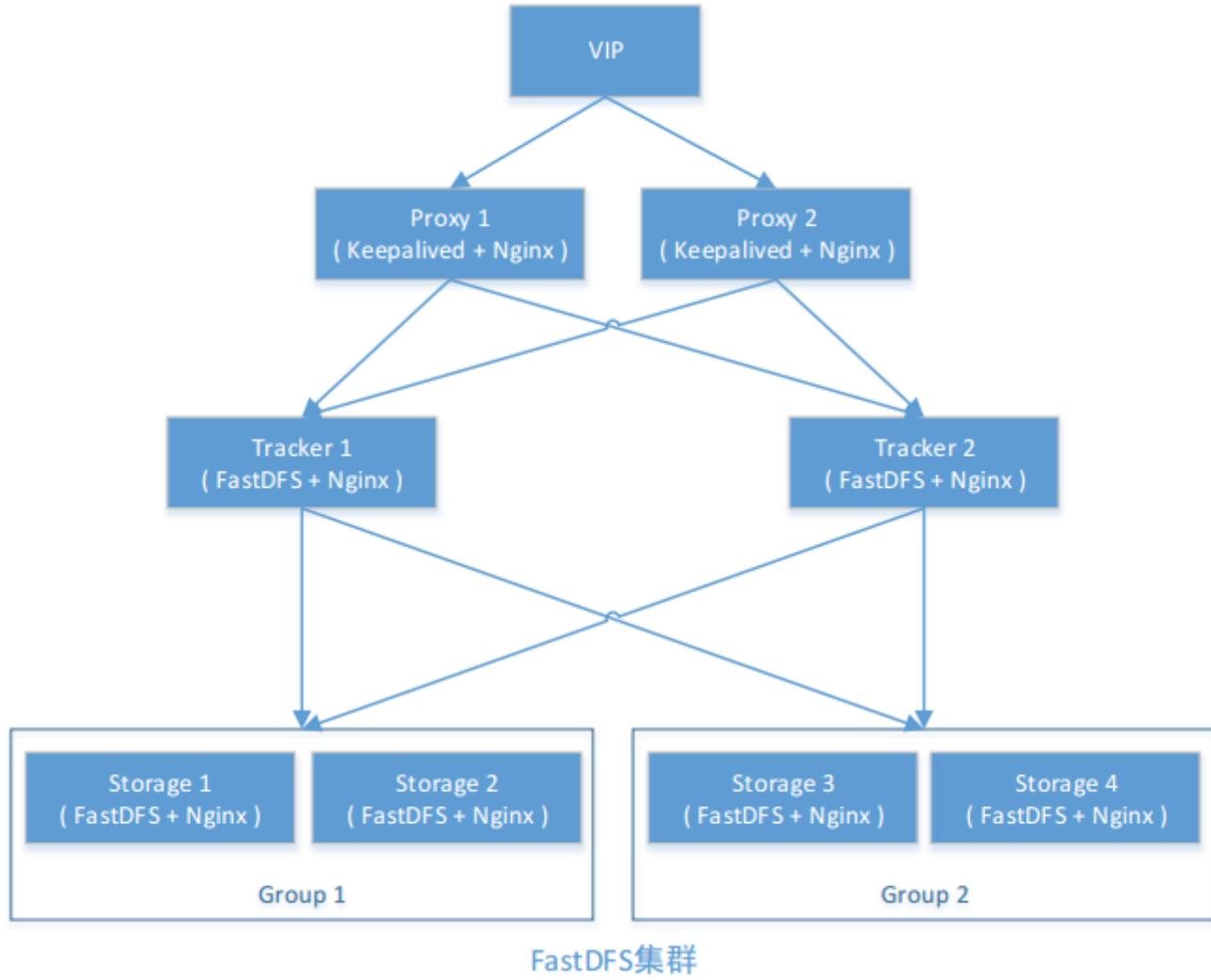
Ceph

FastDFS

FastDFS 服务端有两个角色：跟踪器（tracker）和存储节点（storage）。跟踪器主要做调度工作，在访问上起负载均衡的作用。存储节点存储文件，完成文件管理的所有功能。

Keepalived+Nginx连接多个跟踪器，实现和Web一样的高可用。直接访问这个VIP，获取文件。上传下载也直接写这个VIP。

跟踪器和存储节点都可以由一台或多台服务器构成。跟踪器和存储节点中的服务器均可以随时增加或下线而不会影响线上服务。其中跟踪器中的所有服务器都是对等的，可以根据服务器的压力情况随时增加或减少。各个存储节点上安装Nginx可以提供Web直接访问文件。哪个及其，就哪个IP访问。或者直接写VIP。

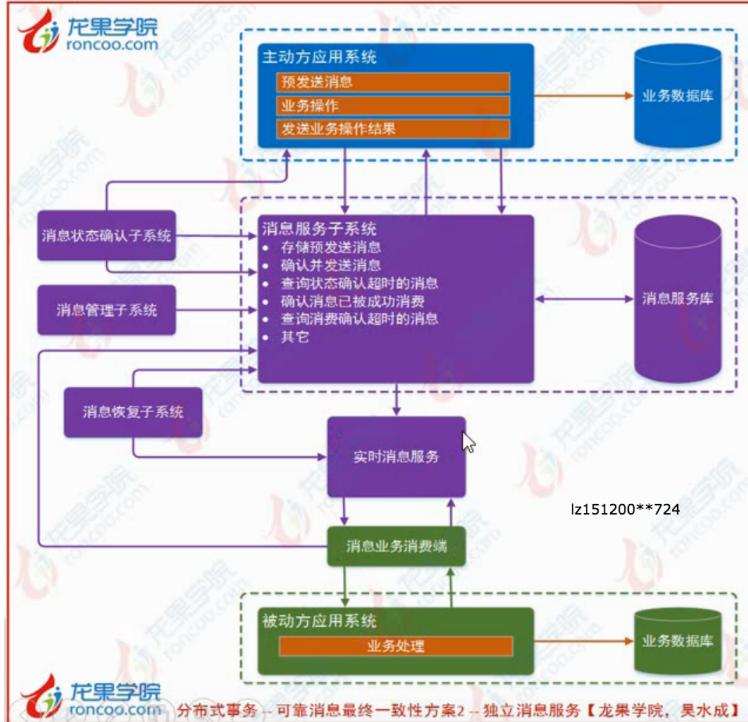


分布式事务

分布式事务解决方案：消息最终一致性（异步确保性）、最大努力通知、TCC事务补偿性（也属于两阶段型）。不管哪种方案，都要求实现幂等性（每次执行结果都相同）和可查询性（可查询这个唯一服务）。

消息最终一致性

可靠消息服务（独立消息服务）的设计



- 1、消息服务子系统；
- 2、消息管理子系统；
- 3、消息状态确认子系统；
- 4、消息恢复子系统；
- 5、实时消息服务；

超级教程系列
《微服务架构的分布式事务解决方案》

我们可以把上面紫色那部分称为可靠的消息服务来实现可靠消息的投递。

对于每块虚框部分，都是一个独立的事务域。

我们从消息最终一致性的四个维护分析：

1. 从消息发送一致性的正向流程分析，我们还是以支付订单，收到银行订单通知后和会计系统记账为例，我们通过消息实现异步的可靠一致性。

在主动方应用系统中，由于预发送消息、业务操作、发送业务操作结果在同一个本地域事务中，在更新订单之前，我先预发送一个消息，发送会计记账凭证，在消息服务子系统中，先存储预发送消息，状态为“预发送”，但这个消息还不能被消费，如果存储域发送消息失败，业务也不会继续执行，直接回滚。如果存储预发送消息成功，返回成果标记给主动方，主动方继续业务操作，执行完业务操作后，又发送业务操作结果，这时在消息服务子系统中，调确认并发送消息接口，把刚才预存储的消息状态改为“可发送”，并把消息扔给实时消息系统MQ中，接着就被消息业务消费端实时的监听并被消费掉。

2. 我们看消息发送一致性的异常处理流程：主动方发给消息服务子系统预发送消息，如果消息服务子系统出错，则会返回出错，主动方不会继续执行业务，而且还会回滚。如果主动方业务操作完毕后，发送业务操作结果网络中断，或业务操作失败，消息服务子系统未收到确认并发送消息，或者这个消息一直是一直是发送中，消息服务子系统发送给实时消息服务MQ失败，这时消息确认状态子系统就起作用了，消息确认状态子系统其实就是一个定时任务，定时轮询，定时请求主动方应用，查询消息数据库中状态确认超时的消息，拿出请求主动方，主动方应该提供对应的业务查询接口，提供业务是否被处理成功。如果是处理成功了，消息状态确认子系统会确认消息，修改消息状态为“可发送”，并发送消息给实时消息系统MQ，被消费掉。如果业务返回的是业务没有被处理成功，消息状态确认子系统会调用消息服务子系统去删除这条“预发送”消息。

3. 我们看消息消费的正向流程：投递进实时消息服务MQ的消息，会被监听这个队列的消息业务消费端监听到，当监听到后，会调被动方应用服务，被动方在本地进行业务处理，业务处理一旦成功，会返回一个结果，是成功或失败，消息业务消费端收到返回状态后，会给实时消息服务MQ发送一个MQ的ACK，从队列里面删除这条消息。同时会调消息服务子系统中的确认消息已被成功消费接口，确认后，消息服务子系统会从消息服务中将这条消息删除掉，或修改状态为“已消费”。

4. 我们看消息消费的异常处理流程：消息业务消费端和被动方应用系统可能是网络通信，在消费端可能会存在网络问题，导致消息业务消费端不能请求MQ的ACK，或调确认消息已被成功消费接口，这时消息恢复子系统就起作用了，它定时的查询消息确认超时的消息，拿出来，把它重新的投递到实时消息服务中让重做，消费端是要实现幂等性的，重做多次的结果都一样，直到消费端返回成功接口。

另外可以提供一个消息管理子系统WebUI，提供对积累消息的查看，手动管理。

这个方案的优点：

- 1、消息服务独立部署、独立维护、独立伸缩；
- 2、消息存储可以按需选择不同的数据库来集成实现；
- 3、消息服务可以被相同的使用场景公用，降低重复建设消息服务的成本；
- 4、从应用（分布式服务）设计开发的角度实现了消息数据的可靠性，消息数据的可靠性不依赖于MQ中间件，弱化了对MQ中间件特性的依赖；
- 5、降低了业务系统与消息系统间的耦合，有利于系统的扩展维护；图中紫色部分是独立的，业务只需要提供一个查询业务执行状态接口。

弊端/局限：

- 1、一次消息发送需要两次请求；
- 2、主动方应用系统需要实现业务操作状态校验查询接口。

我们只需要MQ的消息队列功能，不需要配置MQ重试。因为我们的服务会自己重试。

最大努力通知

最大努力通知：

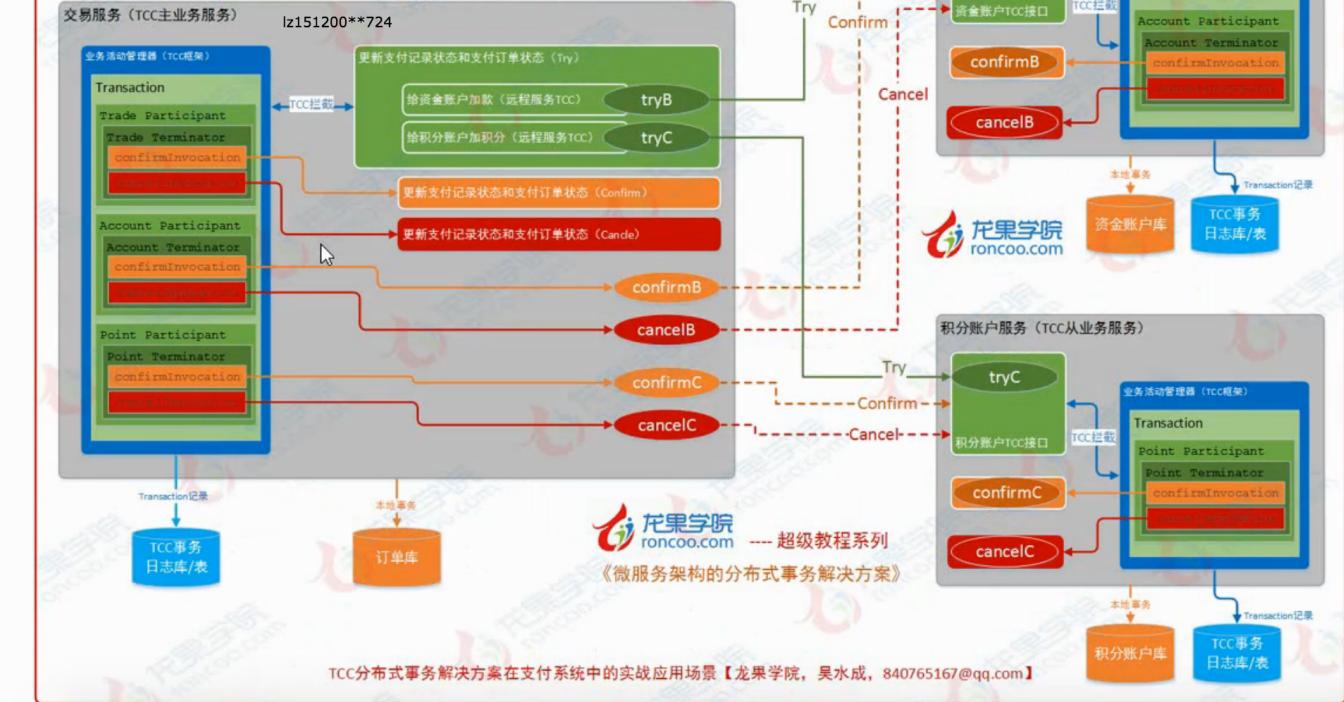
1. 业务活动的主动方在完成业务处理后，向消息服务队列中发送通知消息；
2. 接收消息服务端在收到MQ消息以后，它会在通知接收端失败后，还重复的再投递消息，最大努力的保证通知成功。而不是收到消息以后，只通知一次。接收消息服务端使用了一个叫DeployQueue，在一个这个队列中，每个对象都有一个延时时间，当哪个的延时时间到了后它才会出队列。

TCC事务补偿

TCC适用范围：

- 强隔离性、严格一致性要求的业务活动；
- 适用于执行时间较短的业务（比如处理账户、收费等业务），要求实时性比较高的业务。

TCC中间件+业务配合使用流程：



1. 进订单服务主TRY方法前，Compensable拦截，ROOT类型，生成一个Transaction（类型为主支），又被TransactionContext拦截，ROOT类型，生一个新的TransactionXid xid, xid两个属性，参数1为食物TransactionID，参数2为UUID branchQualifier字符串，获取confirm方法名和cancel方法名，构建ConfirmInvocation和CancelInvocation，使用confirmInvo和CancelInvo构成Terminal，Terminal和xid构成一个Participant放到Transaction中。开始执行TRY方法
2. 消费方法被TransactionContext拦截，CONSUMER类型，获取当前Transaction，生一个新的TransactionXid xid, xid两个属性，参数1为食物TransactionID，参数2为UUID branchQualifier字符串，生TransactionContext对象（参数1传xid，参数2传当前事务状态为TRYING），构建ConfirmInvocation和CancelInvocation，confirm和cancel方法的参数1是TransactionContext对象（参数1传xid，参数2传事务状态分别为CONFIRMING和CANCELING）。方法名都为账户服务的TRY方法名，使用confirmInvo和CancelInvo构成Terminal，Terminal和xid构成一个Participant放到Transaction中。开始执行TRY方法。远程调用资金账户服务
3. 进资金账户服务从TRY方法前，Compensable拦截，PROVIDER类型，获取当前的transactionContext保存的事务状态，当前为TRYING状态，生成一个Transaction（类型为分支），又被TransactionContext拦截，PROVIDER类型，生一个新的TransactionXid xid, xid两个属性，参数1为食物TransactionID，参数2为UUID branchQualifier字符串，获取confirm方法名和cancel方法名，构建ConfirmInvocation和CancelInvocation，使用confirmInvo和CancelInvo构成Terminal，Terminal和xid构成一个Participant放到Transaction中。开始执行TRY方法
4. 又开始积分的消费方法，完了后，主中一个Transaction，里包含3个Participant。Transaction的状态为TRYING，积分服务和资金服务各包含一个Transaction，里包含1个Participant。Transaction的状态为TRYING。
5. 如果上面3个TRY任意有失败，或者成功

```

1. private Object rootMethodProceed(ProceedingJoinPoint pjp) throws Throwable {
2.     logger.debug("==>rootMethodProceed");
3.     transactionConfigurator.getTransactionManager().begin(); // 事务开始（创建事务日志记录，并
4.     Object returnValue = null; // 返回值
5.     try {
6.
7.         logger.debug("==>rootMethodProceed try begin");
8.         returnValue = pjp.proceed(); // Try（开始执行被拦截的方法）
9.         logger.debug("==>rootMethodProceed try end");

```

```

10.
11.     } catch (OptimisticLockException e) {
12.         logger.warn("==>compensable transaction trying exception.", e);
13.         throw e; //do not rollback, waiting for recovery job
14.     } catch (Throwable tryingException) {
15.         logger.warn("compensable transaction trying failed.", tryingException);
16.         transactionConfigurator.getTransactionManager().rollback();
17.         throw tryingException;
18.     }
19.     logger.info("==>rootMethodProceed begin commit()");
20.     transactionConfigurator.getTransactionManager().commit(); // Try检验正常后提交(事务管理器
21.     return returnValue;
22. }

```

rollback和commit方法差不多一样。我们现在就说执行commit方法

6. 主的commit方法中

```

1. public void commit() {
2.     LOG.debug("==>TransactionManager commit()");
3.     Transaction transaction = getCurrentTransaction();
4.     transaction.changeStatus(TransactionStatus.CONFIRMING);
5.     LOG.debug("==>update transaction status to CONFIRMING");
6.     transactionConfigurator.getTransactionRepository().update(transaction);
7.     try {
8.         LOG.info("==>transaction begin commit()");
9.         transaction.commit(); //阻塞, 3个Participant
10.        transactionConfigurator.getTransactionRepository().delete(transaction);
11.    } catch (Throwable commitException) {
12.        LOG.error("compensable transaction confirm failed.", commitException);
13.        throw new ConfirmingException(commitException);
14.    }
15. }

```

修改事务Transaction状态为CONFIRMING，然后阻塞执行事务的commit。执行完commit后，才删除这个事务。

阻塞的过程都干了什么

7. 循环执行每个Participant的commit方法，这个方法里会动态的执行每个Confirm方法。

8. 执行主的confirm方法，如果抛出异常了，直接抛出一场，然后进上面第6步的Throwable commitException，如果成功了，接着第二个Participant

9. 进入资金账户的TRY方法，参数1的TransactionContext的事务类型为CONFIRMING。被Compensable拦截，PROVIDER类型，transactionContext类型为CONFIRMING，取出当前的事务Transaction，修改事务状态为CONFIRMING，然后执行

```
transactionConfigurator.getTransactionManager().commit();
```

这个执行这个事务里的所有Participant，执行真正的confirm方法，

10.

```

1. public void commit() {
2.     LOG.debug("==>TransactionManager commit()");
3.     Transaction transaction = getCurrentTransaction();
4.     transaction.changeStatus(TransactionStatus.CONFIRMING);
5.     LOG.debug("==>update transaction status to CONFIRMING");
6.     transactionConfigurator.getTransactionRepository().update(transaction);
7.     try {
8.         LOG.info("==>transaction begin commit()");
9.         transaction.commit();
10.        transactionConfigurator.getTransactionRepository().delete(transaction);
11.    } catch (Throwable commitException) {

```

```
12.     LOG.error("compensable transaction confirm failed.", commitException);
13.     throw new ConfirmingException(commitException);
14. }
15. }
```

执行完后，删除这个Transaction

11. 继续积分服务，和第9步一样

12. 接着第6步的commit完后的删除这个Transaction。

分布式配置

我们以前使用maven profile里配置，现在跨语言了，基于disconf。

同一个上线包，无须改动配置，即可在多个环境中(RD/QA/PRODUCTION) 上线

更改配置，无需重新打包或重启，即可实时生效

提供web平台，统一管理 多个环境(RD/QA/PRODUCTION)、多个产品 的所有配置

分布式跟踪

Zipkin 是一款开源的分布式实时数据追踪系统（Distributed Tracking System），由 Twitter 开发。其主要功能是聚集来自各个异构系统的实时监控数据，用来追踪微服务架构下的系统延时问题。

Zipkin 以 Trace 结构表示对一次请求的追踪，又把每个 Trace 拆分为若干个有依赖关系的 Span。在微服务架构中，一次用户请求可能会由后台若干个服务负责处理，那么每个处理请求的服务就可以理解为一个 Span（可以包括 API 服务，缓存服务，数据库服务以及报表服务等）。当然这个服务也可能继续请求其他的服务，因此 Span 是一个树形结构，以体现服务之间的调用关系。

Zipkin 的用户界面除了可以查看 Span 的依赖关系之外，还以瀑布图的形式显示了每个 Span 的耗时情况，可以一目了然的看到各个服务的性能状况。打开每个 Span，还有更详细的数据以键值对的形式呈现，而且这些数据可以在装备应用的时候自行添加。

Zipkin 主要由四部分构成：收集器、数据存储、查询以及 Web 界面。Zipkin 的收集器负责将各系统报告过来的追踪数据进行接收；而数据存储默认使用 Cassandra，也可以替换为 MySQL；查询服务用来向其他服务提供数据查询的能力，而 Web 服务是官方默认提供的一个图形用户界面。而各个异构的系统服务向 Zipkin 报告数据。

Brave 是用来装备 Java 程序的类库，提供了面向 Standard Servlet、Spring MVC、Http Client、JAX RS、Jersey、Resteasy 和 MySQL、JDBC 等接口的装备能力，可以通过编写简单的配置和代码，让基于这些框架构建的应用可以向 Zipkin 报告数据。同时 Brave 也提供了非常简单且标准化的接口，在以上封装无法满足要求的时候可以方便扩展与定制。

对于其他语言，都有对应的Zipkin装备插件。

LVS、Nginx、Haproxy、Keepalived

计算机集群分类

计算机集群架构按照功能和结构一般分成以下几类：

1) 负载均衡集群 (Loadbalancingclusters) 简称LBC

负载均衡运行时，一般通过一个或多个前端负载均衡器将客户访问请求分发到后端一组服务器上，从而达到整个系统的高性能和高可用性。如LVS、Haproxy、Nginx。

2) 高可用性集群 (High-availabilityclusters) 简称HAC

指当集群中的任意一个节点失效的情况下，节点上的所有任务自动转移到其他正常的节点上，并且此过程不影响整个集群的运行，不影响业务的提供。如keepalived

3) 高性能计算集群 (High-perfomanceclusters) 简称HPC

高性能计算集群采用将计算任务分配到集群的不同计算节点提高计算能力。

负载均衡技术有很多实现方案，如基于DNS域名轮流解析的方法、基于客户端调度访问的方法、基于应用层系统负载的调度方法，还有基于IP地址的调度方法，在这些负载调度算法中，执行效率最高的就是IP负载均衡技术。

NAT/SNAT/DNAT

NAT (Network Address Translation, 网络地址转换) 是将IP数据包头中的IP地址转换为另一个IP地址的过程。在实际应用中，NAT主要用于实现私有网络访问公共网络的功能。这种通过使用少量的公有IP地址代表较多的私有IP地址的方式，将有助于减缓可用IP地址空间的枯竭。

DNAT Destination Network Address Translation 目的网络地址转换。

SNAT Source Network Address Translation 源网络地址转换，其作用是将ip数据包的源地址转换成另外一个地址。

为什么要进行ip地址转换啊？我们要看一下局域网用户上公网的原理：

内部地址要访问公网上的服务时（如web访问），内部地址会主动发起连接，由路由器或者防火墙上的网关对内部地址做个地址转换，将内部地址的私有IP转换为公网的公有IP，网关的这个地址转换称为SNAT，主要用于内部共享IP访问外部。

当内部需要提供对外服务时（如对外发布web网站），外部地址发起主动连接，由路由器或者防火墙上的网关接收这个连接，然后将连接转换到内部，此过程是由带有公网IP的网关替代内部服务来接收外部的连接，然后在内部做地址转换，此转换称为DNAT，主要用于内部服务对外发布。

LVS

LVS的IP负载均衡技术是通过ipvs内核模块来实现的，ipvs是LVS集群系统的核心软件，它的主要作用是：安装在Director Server上，同时在Director Server上虚拟出一个IP地址，用户必须通过这个虚拟的IP地址访问集群服务。这个虚拟IP一般称为LVS的VIP，即Virtual IP。访问的请求首先经过VIP到达负载调度器，然后由负载调度器从Real Server列表中选取一个服务节点响应用户的请求。当用户的请求到达负载调度器后，调度器如何将请求发送到提供服务的Real Server节点，而Real Server节点如何返回数据给用户，是ipvs实现的重点技术。

ipvs实现负载均衡机制有四种，分别是NAT、TUN、DR、以及后来经淘宝开发的FullNat。下面将详细介绍这四种机制的工作模型。

LVS分为两个部件：ipvs和ipvsadm

ipvsadm是给用户使用的工具，有它给在内核工作的ipvs分配策略。

Director Server在收到客户端请求时，会基于调度算法从Real Server中选择一个响应客户机的请求。10种调度算法：

- 1) RR 轮询(Round Robin) 无需记录所有连接状态，新的连接请求被轮流分配至各RealServer；
- 2) WRR 加权轮询(Weighted RR) 无需记录所有连接状态，通过设定一定的权重值来分配连接请求；
- 3) SH 源地址哈希(Source Hashing) 通过一个散列函数将去往同一个目的IP的请求映射到一台服务器或链路上；
- 4) DH 目标地址哈希(Destination Hashing) 通过一个散列函数将来自同一个源IP的请求映射到一台服务器或链路上；
- 5) LC 最少连接数(Least Connection) 根据当前各服务器的连接数来估计服务器的负载情况，把新的连接分配给连接数最小的服务器；
- 6) WLC 加权最少连接数(Weighted LC) 与LC类似，根据当前各服务器的连接数来估计服务器的负载情况，把新的连接分配给连接数最小的服务器；
- 7) SED 最短期望延迟(Shortest Expect Delay) 这个算法主要是优化LC的，在服务均在请求少的时候避免负载到一台服务器上做的优化；
- 8) NQ 永不排队(Nerver Queue) 在负载低时，请求直接分配到空闲服务器上，不会产生请求等待；当服务器都很忙时，将轮询；
- 9) LBLC 基于本地最少连接 (Locality-Based Least Connection) 根据请求的目标IP地址找出该目标IP地址最近使用的RealServer，若该Real Server是可用的且没有超载，将请求发送到该服务器；若服务器不存在，或者该服务器超载且有服务器处于一半的工作负载，则用“最少链接”的原则选出一个可用的服务器，将请求发送到该服务器。
- 10) LBLCR 带复制的基于本地最少连接(Replicated and Locality-Based Least Connection) 根据请求的目标IP地址找出该目标IP地址对应的服务器组，按“最小连接”原则从服务器组中选出一台服务器，若服务器没有超载，将请求发送到该服务器；若服务器超载，则按“最小连接”原则从这个集群中选出一台服务器，将该服务器加入到服务器组中，将请求发送到该服务器。

ipvs有如下4种工作模式：

1 NAT

修改请求报文的目标IP,相当于多目标IP的DNAT

DR只需要将VIP配置到DR上，它的工作机制是，将收到的集群服务请求报文目标IP地址转换成经调度算法计算得出的后端主机IP地址，然后后端主机将响应报文发送至DR，再由DR将源地址转换成VIP的地址。

流程：

```
client-ip => direct-ip 80  
client-ip => real-ip 9000 DNAT  
client-ip => real-ip 9000  
direct-ip => client-ip 80 SNAT
```

原理简述：

- 1)客户端请求数据，目标IP为VIP
- 2)请求数据到达LB服务器，LB根据调度算法将目的地址修改为RIP地址及对应端口（此RIP地址是根据调度算法得出的。）并在连接HASH表中记录下这个连接。
- 3)数据包从LB服务器到达RS服务器webserver，然后webserver进行响应。Webserver的网关必须是LB，然后将数据返回给LB服务器。
- 4)收到RS的返回后的数据，根据连接HASH表修改源地址VIP&目标地址CIP，及对应端口80.然后数据就从LB出发到达客户端。
- 5)客户端收到的就只能看到VIP\VIP信息。

NAT模式优缺点：

- 1、NAT技术将请求的报文和响应的报文都需要通过LB进行地址改写，因此网站访问量比较大的时候LB负载均衡调度器有比较大的瓶颈，一般要求最多之能10-20台节点；
- 2、只需要在LB上配置一个公网IP地址就可以了；
- 3、每台内部的节点服务器的网关地址必须是调度器LB的内网地址，RIP和DIP必须在同一网段内；
- 4、NAT模式支持对IP地址和端口进行转换。即用户请求的端口和真实服务器的端口可以不一致。
- 5、效率低

2 DR

操纵封装新的MAC地址（默认）

client向目标vip发出请求，Director Server接收，LVS根据负载均衡算法选择一台Real Server，将IP数据包中的目标mac地址改为Real Server的地址，发送到局域网（IP数据包封装到以太网，送到局域网，先ARP广播活得目标IP，重新封包），Real Server收到这个帧，发现目标IP与它匹配，处理报文，随后重新封装报文，目标IP为client，源IP为它自己Real Server。

流程：

direct-mac + client-ip => direct-ip 80

direct-mac => real-mac 修改mac

real-mac + client-ip => direct-ip 80

real-ip => client-ip

原理简述：

在DR模式中，调度器根据各个真实服务器的负载情况，连接数多少等，动态地选择一台服务器，不修改目标IP地址和目标端口，也不封装IP报文，而是将请求报文的数据帧的目标MAC地址改写成真实服务器的MAC地址。然后再将修改的数据帧在服务器组的局域网上发送。因为数据帧的MAC地址是真实服务器的MAC地址，并且又在同一个局域网。那么根据局域网的通讯原理，真实复位是一定能够收到由LB发出的数据包。真实服务器接收到请求数据包的时候，解开IP包头查看到的目标IP是VIP。（此时只有自己的IP符合目标IP才会接收进来，所以我们需要在本地的回环接口上面配置VIP。另：由于网络接口都会进行ARP广播响应，但集群的其他机器都有这个VIP的lo接口，都响应就会冲突。所以我们需要把真实服务器的lo接口的ARP响应关闭掉。）然后真实服务器做成请求响应，之后根据自己的路由信息将这个响应数据包发送回给客户，并且源IP地址还是VIP。

DR模式优缺点：

- 1、通过在调度器LB上修改数据包的目的MAC地址实现转发。注意源地址仍然是CIP，目的地址仍然是VIP地址。
- 2、请求的报文经过调度器，而RS响应处理后的报文无需经过调度器LB，因此并发访问量大时使用效率很高（和NAT模式比）
- 3、因为DR模式是通过MAC地址改写机制实现转发，因此所有RS节点和调度器LB只能在一个局域网里面
- 4、RS主机需要绑定VIP地址在LO接口上，并且需要配置ARP抑制。
- 5、RS节点的默认网关不需要配置成LB，而是直接配置为上级路由的网关，能让RS直接出网就可以。
- 6、由于DR模式的调度器仅做MAC地址的改写，所以调度器LB就不能改写目标端口，那么RS服务器就得使用和VIP相同的端口提供服务。
- 7、RS跟DR要在同一物理网络内（不能由路由器分隔）；
- 8、请求报文经过DR，但响应报文一定不经过DR
- 9、不支持端口映射；
- 10、性能最高，但不能跨域LAN

3 TUN

在原请求IP报文之外新加一个IP首部，ip隧道

在数据包必须传递到另一个网络或因特网上时，可以使用ip隧道，ip隧道能够将数据包从一个子网或虚拟局域网（VLAN）转发到另一个子网或虚拟局域网（VLAN）。

TUN转发方法允许你将集群节点放在与Director不同的网络上。

DR是直接在数据包中直接添加IP首部（源IP:DR和目标IP:RS），这样就是隧道技术。

流程：

client-ip => direct-ip 80

ip-tunnel client-ip => direct-ip

real-ip => client-ip

原理简述：

1)客户请求数据包，目标地址VIP发送到LB上。

2)LB接收到客户请求包，进行IP Tunnel封装。即在原有的包头加上IP Tunnel的包头。然后发送出去。

3)RS节点服务器根据IP Tunnel包头信息（一种逻辑上的隐形隧道，只有LB和RS之间懂）收到请求包，然后解开IP Tunnel包头信息，得到客户的请求包并进行响应处理。

4)响应处理完毕之后，RS服务器使用自己的出公网的线路，将这个响应数据包发送给客户端。源IP地址还是VIP地址。

tun模式优缺点：

- 1、RIP、DIP、VIP都得是公网地址；
- 2、RS的网关不会指向也不可能指向DIP；
- 3、请求报文经过DR，但响应报文一定不经过DR；
- 4、不支持端口映射；
- 5、RS的OS必须得支持隧道功能；
- 6、需要隧道支持

4 Fullnat

同时修改请求报文的源和目标IP

Fullnat是淘宝开源的一种lvs转发模式，主要思想：引入local address（内网ip地址），cip-vip转换为lip->rip，而lip和rip均为IDC内网ip，可以跨vlan通讯，这刚好符合我们的需求，因为我们的内网是划分了vlan的。

FULLNAT模式要重编LVS机器内核。

client-ip => direct-ip 80

lvs local-ip => real-ip SNAT + DNAT

real-ip =>lvs local-ip

direct-ip => client-ip SNAT + DNAT

Nginx

NGINX以高性能的负载均衡器，缓存，和web服务器闻名，相较于Apache、lighttpd具有占有内存少，稳定性高等优势。nginx不采用每客户机一线程的设计模型，而是充分使用异步逻辑，削减了上下文调度开销，所以并发服务能力更强。整体采用模块化设计，有丰富的模块库和第三方模块库，配置灵活。在Linux操作系统下，nginx使用epoll事件模型，得益于此，nginx在linux操作系统下效率相当高。

Nginx内核

Nginx由内核和模块组成，其中，内核的设计简洁，完成的工作也非常简单，仅仅通过查找配置文件将客户端请求映射到一个location block（location是Nginx配置中的一个指令，用于URL匹配），而在该location中所配置的每个指令将会启动不同的模块去完成相应的工作。

Nginx的模块从结构上分为核心模块、基础模块和第三方模块：

- 核心模块：HTTP模块、EVENT模块和MAIL模块
- 基础模块：HTTP Access模块、HTTP FastCGI模块、HTTP Proxy模块和HTTP Rewrite模块，
- 第三方模块：HTTP Upstream Request Hash模块、Notice模块和HTTP Access Key模块。

用户根据自己的需要开发的模块都属于第三方模块。正是有了这么多模块的支撑，Nginx的功能才会如此强大。

Nginx的模块直接被编译进Nginx，因此属于静态编译方式。启动Nginx后，Nginx的模块被自动加载，不像Apache，首先将模块编译为一个so文件，然后在配置文件中指定是否进行加载。在解析配置文件时，Nginx的每个模块都有可能去处理某个请求，但是同一个处理请求只能由一个模块来完成。

Nginx的进程模型

在工作方式上，Nginx分为单工作进程和多工作进程两种模式。

- 在单工作进程模式下，除主进程外，还有一个工作进程，工作进程是单线程的；
- 在多工作进程模式下，每个工作进程包含多个线程。

Nginx默认为单工作进程模式。

Nginx在启动后，会有一个master进程和多个worker进程。

- master进程，主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。我们要控制nginx，只需要通过kill向master进程发送信号就行了。
- worker进程，都会被master fork创建出来的，基本的网络事件，则是放在worker进程中来处理了。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程。worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致。

这里面的原因与nginx的进程模型以及事件处理模型是分不开的。在master里面，先建立需要listen的socket (listenfd)，然后再fork出多个worker进程。当用户请求nginx服务的时候，每个worker的listenfd变的可读，并且这些worker会抢一个叫accept_mutex的东西，accept_mutex是互斥的，一个worker得到了，其他的worker就歇菜了。而抢到这个accept_mutex的worker就开始“读取请求--解析请求--处理请求”，数据彻底返回客户端之后（目标网页出现在电脑屏幕上），这个事件就算彻底结束。

nginx底下的worker进程抢注用户的要求，同时搭配“异步非阻塞”的方式，实现高并发量。

在nginx根部设置异步非阻塞

```
1. #工作模式及每个进程连接数上限
2. events {
3.     use epoll;
4.     worker_connections 1024;      #所以nginx支持的总连接数就等于worker_processes * worker_conn
5. }
```

常见配置：

1. 编写一个Nginx的access模块，要求准许192.168.3.29/24的机器访问，准许10.1.20.6/16这个网段的所有机器访问，准许34.26.157.0/24这个网段访问,除此之外的机器不准许访问。

location{/

```
access 192.168.3.29/24;
access 10.1.20.6/16;
access 34.26.157.0/24;
```

```
deny all;  
}
```

2. Nginx实现多Tomcat负载均衡

Tomcat服务

```
192.168.1.177:8001
```

```
192.168.1.177:8002
```

```
192.168.1.177:8003
```

Nginx配置

```
upstream mytomcats {  
    server 192.168.1.177:8001;  
    server 192.168.1.177:8002;  
    server 192.168.1.177:8003;  
}  
  
server {  
    listen 80;  
    server_name www.iu14.com;  
    location ~* \.(jpg|gif|png|swf|flv|wma|wmv|asf|mp3|mmf|zip|rar)$ {  
        root /web/www/html/;  
    }  
    location / {  
        proxy_pass http://mytomcats;  
        proxy_redirect off;  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
        client_max_body_size 10m;  
        client_body_buffer_size 128k;  
        proxy_connect_timeout 90;  
        proxy_send_timeout 90;  
        proxy_read_timeout 90;  
        proxy_buffer_size 4k;  
        proxy_buffers 4 32k;  
        proxy_busy_buffers_size 64k;  
        proxy_temp_file_write_size 64k;  
    }  
}
```

upstream指定负载均衡组，指定其Tomcat成员

location ~* \.(jpg|gif|.....实现了静态资源分离。ps：在location指令使用正则表达式后再用alias指令，Nginx是不支持的。

3. Nginx实现静态资源分离

Tomcat服务

```
192.168.1.177:8000
```

Nginx配置

```
server {  
    listen 80;  
    server_name www.iu14.com;
```

```
root /web/www/html;
location /img/ {
alias /web/www/html/img/;
}
location ~ (.jsp)|(.do)$ {
proxy_pass http://192.168.1.177:8000;
proxy_redirect off;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    client_max_body_size 10m;
    client_body_buffer_size 128k;
    proxy_connect_timeout 90;
proxy_send_timeout 90;
proxy_read_timeout 90;
    proxy_buffer_size 4k;
    proxy_buffers 4 32k;
    proxy_busy_buffers_size 64k;
    proxy_temp_file_write_size 64k;
}
}
```

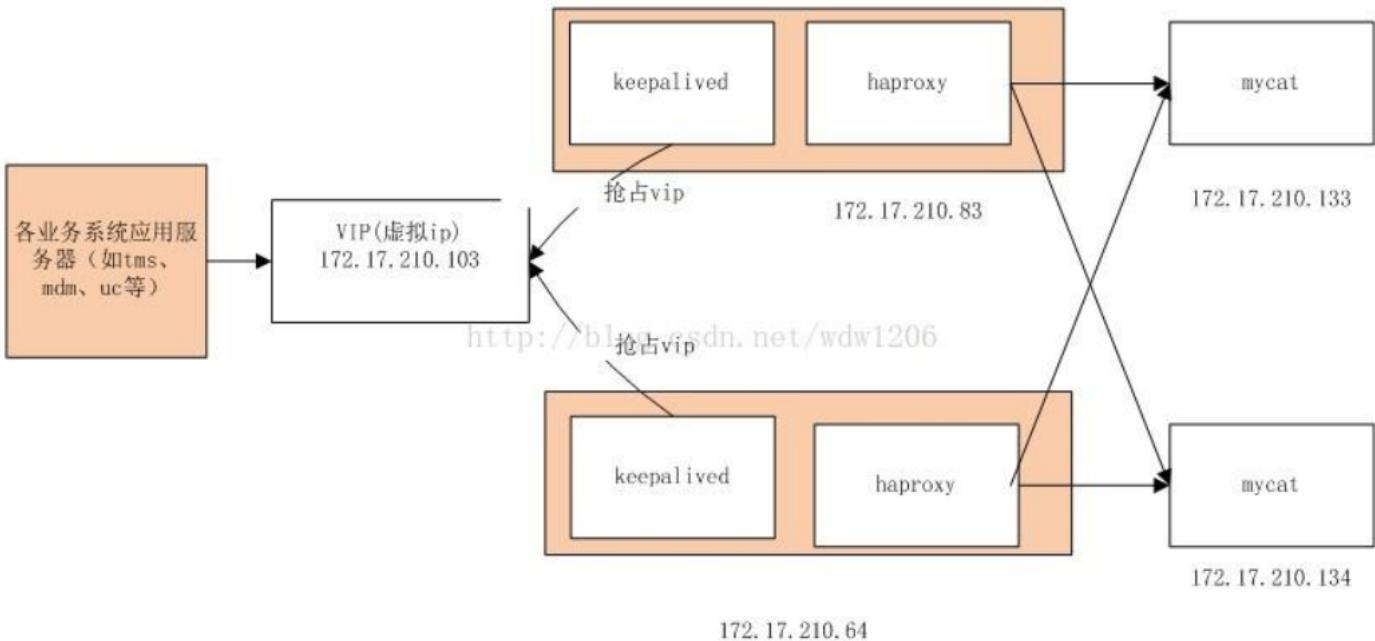
第一个location指令将/web/www/html/img/目录下的静态文件交给Nginx来完成。最后一个location指令将所有以.jsp、.do结尾的文件都交给Tomcat服务器的8080端口来处理。

haproxy

LVS是基于Linux ipvs实现的一种软负载，HAProxy是开源的并且基于第三应用实现的软负载。HAProxy相比LVS的使用要简单很多，功能也丰富。当前，HAProxy支持两种主要的代理模式：四层的TCP和七层HTTP。在4层模式下，HAProxy仅在客户端和服务器之间转发双向流量。7层模式下，HAProxy会分析协议，并且能通过允许、拒绝、交换、增加、修改或者删除请求(request)或者回应(response)里指定内容来控制协议，这种操作要基于特定规则。

HAProxy主要功能就是负载均衡，负载均衡算法支持很多，支持健康检查，支持各种会话保持方式，支持远程信息获取等。

HAProxy可以作为MySQL、邮件或其它的非web的负载均衡，我们常用于它作为mysql(读)负载均衡；
HAProxy的负载均衡能力虽不如LVS，但也是相当不错，但提供了七层工作，LVS没有。



LVS、Nginx、HAProxy对比

Nginx优点：

- 1、工作在网络的7层之上，可以针对http应用做一些分流的策略，比如针对域名、目录结构，它的正则规则比HAProxy更为强大和灵活，这也是它目前广泛流行的主要原因之一。
- 2、Nginx对网络稳定性的依赖非常小，理论上能ping通就能进行负载功能，这个也是它的优势之一；相反LVS对网络稳定性依赖比较大，这点本人深有体会；
- 3、Nginx安装和配置比较简单，测试起来比较方便，它基本能把错误用日志打印出来。LVS的配置、测试就要花比较长的时间了，LVS对网络依赖比较大。
- 4、Nginx可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持url来检测。比如用户正在上传一个文件，而处理该上传的节点刚好在上传过程中出现故障，Nginx会把上传切到另一台服务器重新处理，而LVS就直接断掉了，如果是上传一个很大的文件或者很重要的文件的话，用户可能会因此而不满。
- 5、Nginx不仅仅是一款负载均衡器/反向代理软件，它同时也是功能强大的Web应用服务器。
- 6、Nginx现在作为Web反向加速缓存越来越成熟了，速度比传统的Squid服务器更快。
- 7、Nginx也可作为静态网页和图片服务器，这方面的性能也无对手。还有Nginx第三方模块也很多。

Nginx缺点：

- 1、Nginx仅能支持http、https和Email协议，这个是它的缺点。
- 2、对后端服务器的健康检查，只支持通过端口来检测，不支持通过url来检测。
- 3、不支持Session的直接保持，但能通过ip_hash来解决。

LVS优点：

- 1、抗负载能力强、是工作在网络4层之上仅作分发之用，没有流量的产生，这个特点也决定了它在负载均衡软件里的性能最强的，对内存和cpu资源消耗比较低。
- 2、配置性比较低，这是一个缺点也是一个优点。
- 3、工作稳定，因为其本身抗负载能力很强，自身有完整的双机热备方案，如LVS+Keepalived，不过我们在项目实施中用得最多的还是LVS/DR+Keepalived。
- 4、无流量，LVS只分发请求，而流量并不从它本身出去。

LVS缺点：

1、不支持正则表达式处理，不能做动静分离；而现在许多网站在这方面都有较强的需求，这个是Nginx/HAPerxy+Keepalived的优势所在。

Haproxy优点：

1、HAProxy的优点能够补充Nginx的一些缺点，比如支持Session的保持，Cookie的引导；同时支持通过获取指定的url来检测后端服务器的状态。

3、HAProxy跟LVS类似，本身就只是一款负载均衡软件；单纯从效率上来讲HAProxy会比Nginx有更出色的负载均衡速度，在并发处理上也是优于Nginx的。

4、HAProxy支持TCP协议的负载均衡转发，可以对MySQL读进行负载均衡，对后端的MySQL节点进行检测和负载均衡，可以用LVS+Keepalived对MySQL主从做负载均衡。

Haproxy缺点：

haproxy上扩展性很差，添加新功能很费劲

四层和七层负载均衡区别

二层负载均衡会通过一个虚拟MAC地址接收请求，然后再分配到真实的MAC地址；

三层负载均衡会通过一个虚拟IP地址接收请求，然后再分配到真实的IP地址；

四层通过虚拟IP+端口接收请求，然后再分配到真实的服务器；

七层通过虚拟的URL或主机名接收请求，然后再分配到真实的服务器。

负载均衡器通常称为四层交换机或七层交换机。四层交换机主要分析IP层及TCP/UDP层，实现四层流量负载均衡。七层交换机除了支持四层负载均衡以外，还有分析应用层的信息，如HTTP协议URI或Cookie信息。

负载均衡分为L4 switch（四层交换），即在OSI第4层工作，就是TCP层啦。此种Load Balance不理解应用协议（如HTTP/FTP/MySQL等等）。例子：LVS，F5。

另一种叫做L7 switch（七层交换），OSI的最高层，应用层。此时，该Load Balancer能理解应用协议。例子：haproxy，MySQL Proxy。

最终形成比较理想的基本架构为：Array/LVS – Nginx/Haproxy – Squid/Varnish – AppServer。

Keepalived

VRRP协议产生

对于局域网用户来说，能够时刻与外部网络保持联系是非常重要的。通常情况下，内部网络中的所有主机都设置一条相同的缺省路由，指向出口网关，实现主机与外部网络的通信。局域网客户端判定哪个路由器应该为其到达目标主机的下一跳网关的方式有动态及静态决策两种方式，动态决策方式如Proxy ARP，客户端使用ARP协议获取其想要到达的目标，而后，由某路由以其MAC地址响应此ARP请求；动态路由发现协议的不足之处在于它会导致在客户端引起一定的配置和处理方面的开销，并且，如果路由器故障，切换至其它路由器的过程会比较慢。解决此类问题的一个方案是为客户端静态配置默认路由设备，这大大简化了客户端的处理过程，但也会带来单点故障类的问题。于是就有了VRRP协议。

VRRP虚拟路由冗余协议，来解决局域网主机访问外部网络的可靠性问题。VRRP可以通过在一个路由器组(一个VRRP组)之间共享一个虚拟IP(VIP)，此时仅需要客户端以VIP作为其默认网关即可。

可以认为VRRP是实现路由器高可用的协议，即将N台提供相同功能的路由器组成一个路由器组，这个组里面有一个MASTER和多个BACKUP，MASTER上有一个对外提供服务的VIP（该路由器所在局域网内其他机器的默认路由为该VIP），MASTER会发组播，当BACKUP收不到vrrp包时就认为MASTER宕掉了，这时就需要根据VRRP的优先级来选举一个BACKUP当MASTER，及时将业务切换到其它设备，从而保持通讯的连续性和可靠性，消除了静态路由配置的单点故障。VRRP协议仅适用于IPv4。

VRRP协议工作原理

1、一个VRRP路由器有唯一的标识：

VRID，范围为0-255。该路由器对外表现为唯一的虚拟MAC地址，地址的格式为00-00-5E-00-01-[VRID]。主控路由器负责对ARP请求用该MAC地址做应答这样，无论如何切换，保证给终端设备的是唯一一致的IP和MAC地址，减少了切换对终端设备的影响。

2、VRRP控制报文只有一种：

VRRP通告它使用IP多播数据包进行封装，组地址为224.0.0.18，发布范围只限于同一局域网内。这保证了VRID在不同网络中可以重复使用。为了减少网络带宽消耗，只有主控路由器才可以周期性的发送VRRP通告报文，备份路由器在连续三个通告间隔内收不到VRRP或收到优先级为0的通告则启动新一轮VRRP选举。

3、在VRRP路由器组中按优先级选举主控路由器：

VRRP协议中优先级范围是0-255。在主控路由器的选举中，高优先级的虚拟路由器获胜。对于相同优先级的候选路由器，则按照IP地址大小顺序选举。VRRP还提供了优先级抢占策略，如果配置了该策略，高优先级的备份路由器便会剥夺当前低优先级的主控路由器而成为新的主控路由器。

4、为了保证VRRP协议的安全性，提供了两种安全认证措施：

明文认证和IP头认证明文认证方式要求：在加入一个VRRP路由器组时，必须同时提供相同的VRID和明文密码适合于避免在局域网内的配置错误，但不能防止通过网络监听方式获得密码IP头认证的方式提供了更高的安全性，能够防止报文重放和修改等攻击。

Keepalived是基于VRRP协议实现的保证集群高可用的一个服务软件。它的主要功能是实现真机的故障隔离及负载均衡器间的失败切换FailOver，可以防止单点故障。

keepalived 负责为该服务器抢占 vip(虚拟 ip)，抢占到 vip 后，对该主机的访问可以通过原来的 ip 访问，也可以直接通过 vip 访问。

在不指定配置文件位置的状态下—keepalived默认先查找文件 /etc/keepalived/keepalived.conf

keepalived+nginx双击热备+负载均衡keepalive.conf

```
1. ! Configuration File for keepalived
2. #全局定义块主要配置故障发生时的通知对象以及机器标识
3. global_defs {
4.     notification_email {
5.         #acassen@firewall.loc
6.         #failover@firewall.loc
7.         #sysadmin@firewall.loc
8.     }
9.     #notification_email_from Alexandre.Cassen@firewall.loc
10.    #smtp_server 192.168.200.1
11.    #smtp_connect_timeout 30
12.    router_id LVS_DEVEL #标识本节点的字符串，故障发生时，邮件通知会用到
13. }
14. #vrrp_script区域主要用来做健康检查的，当时检查失败时会将vrrp_instance的priority减少相应的值。
15. vrrp_script chk_http_port {
16.     script "</dev/tcp/127.0.0.1/8088"
17.     interval 1
```

```

18.     weight -2
19.
20. }
21. #VRRP实例定义块主要用来定义对外提供服务的VIP区域及其相关属性
22. vrrp_instance VI_1 {
23.     state MASTER  #可以是MASTER或BACKUP, 不过当其他节点keepalived启动时会将priority比较大的节点选
24.     interface eth2  #对外提供服务的网络接口, 用来发VRRP包
25.     virtual_router_id 51  # 取值在0~255之间, 用来区分多个instance的VRRP组播, 同一网段中该值不能重
26.     priority 100  #用来选举master的, 要成为master, 那么这个选项的值最好高于其他机器50个点, 该项取值范
27.     advert_int 1  #发VRRP包的时间间隔, 即多久进行一次master选举, 可以认为是健康查检时间间隔, 单位为秒
28.     authentication {
29.         auth_type PASS
30.         auth_pass 1111
31.     }
32.     virtual_ipaddress {
33.         192.168.232.16  #可以有多个VIP地址, 每个地址占一行, 不需要指定子网掩码, 必须与RealServer上设
34.     }
35.     track_script {
36.         chk_http_port
37.     }
}

```

Haproxy+Keepalived负载均衡

keepalived.conf

```

1. # Configuration File for keepalived
2.
3. global_defs {
4.     router_id LVS_DEVEL
5. }
6.
7.
8. vrrp_script chk_http_port {
9.     script "/etc/keepalived/checkHaproxy.sh"
10.    interval 2
11.    weight 2
12. }
13.
14. vrrp_instance VI_1 {
15.     state MASTER          #从keepalived这里改成BACKUP
16.     interface eth0
17.     virtual_router_id 51
18.     priority 104          #从keepalived这里改成100吧, 只要从比主小就行, 数字是从0~255, 数字
19.     advert_int 1
20.     authentication {
21.         auth_type PASS
22.         auth_pass 1111
23.     }
24.     track_script {
25.         chk_http_port
26.     }
27.     virtual_ipaddress {
28.         192.168.207.141
29.     }
30. }

```

checkHaproxy.sh

```

1.#!/bin/bash
2. A=`ps -C haproxy --no-header | wc -l`
3. if [ $A -eq 0 ];then

```

```

4.          /usr/local/sbin/haproxy -f /etc/haproxy.cfg
5.          echo "Haproxy start"
6.          sleep 3
7.          if [ `ps -C haproxy --no-header | wc -l` -eq 0 ];then
8.              /etc/init.d/keepalived stop
9.              echo "keepalived stop"
10.         fi
11.     fi

```

LVS+Keepalived负载均衡

```

1. ! Configuration File for keepalived
2. global_defs {
3.     notification_email {
4.         root@localhost.localdomain
5.     }
6.     notification_email_from sns-lvs@gmail.com
7.     smtp_server 127.0.0.1
8.     router_id LVS_DEVEL
9. }
10. vrrp_instance VI_1 {
11.     state MASTER           #从keepalived要写成BACKUP
12.     interface eth0
13.     virtual_router_id 51
14.     priority 100          #从keepalived要写成小于100的数，就写成99就可以了，范围是0~255
15.     advert_int 1
16.     authentication {
17.         auth_type PASS
18.         auth_pass 1111
19.     }
20.     virtual_ipaddress {
21.         192.168.207.140
22.     }
23. }
24. virtual_server 192.168.207.140 80 {
25.     delay_loop 6
26.     lb_algo wrr
27.     lb_kind DR
28.     # persistence_timeout 60          #这里就不要开启持久连接了，为了更方便的查看负载均和的效果
29.     protocol TCP
30.     real_server 192.168.207.129 80 {
31.         weight 3
32.         TCP_CHECK {
33.             connect_timeout 10
34.             nb_get_retry 3
35.             delay_before_retry 3
36.             connect_port 80
37.         }
38.     }
39.     real_server 192.168.207.130 80 {
40.         weight 3
41.         TCP_CHECK {
42.             connect_timeout 10
43.             nb_get_retry 3
44.             delay_before_retry 3
45.             connect_port 80
46.         }
47.     }
48. }

```

压测

电商压测、mysql压测不用写了，想看就看《乐视电商压力测试》和《1Nginx_1Mycat_5tomcat_10mysql——下单性能测试》文章。

直接了解清楚现在leengine、matrix自身的能力和支持业务的能力。

DevOps

源代码库：Gitlab

构建代码：CI、Sonar代码质量分析

测试代码：Junit、JMeter

部署代码：SaltStack、Ansible、Vagrant、Docker、Kubernetes

监控代码：ELK、Open Falcon。

白板系统：Benarychat

硬件服务器介绍

机架服务器

机架服务器的主要优势表现在节省空间，机架服务器的宽度为19英寸，高度以U为单位(1U=1.75英寸=44.45毫米)，通常有1U，2U，3U，4U，5U，7U几种标准的服务器。

价格方面，机架服务器一般比同等配置的塔式服务器售价高出30%左右。

刀片服务器

刀片服务器是一种高可用高密度的低成本服务器平台，其主要结构为一大型主体机箱，内部可插上许多“刀片”，其中每一块刀片类似于一个个独立的服务器，它们可以通过本地硬盘启动自己的操作系统。在集群模式下，所有的刀片可以连接起来提供高速的网络环境，共享资源，为相同的用户群服务。根据所需要承担的服务器功能，刀片服务器被分成服务器刀片、网络刀片、存储刀片、管理刀片、光纤通道SAN刀片、扩展I/O刀片等等不同功能的刀片服务器。

服务规格

镜像仓库后端存储

1. 服务特性

网络IO，CPU

单万兆网卡，纯内网

CPU: E5-2620 v3 @ 2.40GHZ

内存：64GB

2. 服务器网卡配置

万兆网卡bond 上联

3. 操作系统配置

CentOS Linux release 7.2.1511 (webserver)

4. raid 配置和分区设置

每台机器配置12块6TB的SATA盘， 两两做raid0

var分区至少50GB

系统盘必须要有两块， 做raid 1

5. 机架部署网络拓扑

四台机器必须位于不同的机架， 做到电源隔离

四台中预留一台同型号的存储机器做备用， 因为集群规模过小， 一旦一台服务器故障， 所有数据均降级， 数据风险极大

1 时间需要同步

2 开通nat

负载均衡nginx机器

1. 服务特性

网络IO, CPU

网卡： 万兆

CPU: 2620*2

2. 服务器网卡配置

两块万兆网卡， 一个公网， 一个内网， 内网网卡需要配置10个内网IP。

3. 操作系统配置

centos 7 (webserver)

4. raid 配置和分区设置

raid1， 默认分区（出去系统空间， 剩余给/data）

5. 机架部署网络拓扑

集群内服务器需要放置在不同机柜， 其他无要求。

1 时间需要同步

2 开通nat

Leengine管理节点

1. 服务特性

cpu, 内存

2. 服务器网卡配置

管理端配置在万兆

3. 操作系统配置

centos7 (webserver)

4. raid 配置和分区设置

A1-3 [CPU:24C (E5-2620V3*2)
内存:128G (16G*8) 硬盘:600G (系统 raid1), 1.5T (数据 raid10)
标配实体机 (普通应用) ||26500]

5. 机架部署网络拓扑

集群内服务器需要放置在不同机柜，其他无要求。

1 时间需要同步

2 开通nat

数据库

1. 服务特性

磁盘io、cpu

美国通用1U机器配置Intel E5-2620V3*2/16G*8/SAS 10K2.5''600G*8/RAID0,1,5,6,10 1G cache

2. 服务器网卡配置

千兆bond/万兆

3. 操作系统配置

centos6

4. raid 配置和分区设置

系统分区raid1

数据分区raid10

5. 机架部署网络拓扑

集群内服务器需要放置在不同机柜，上联交换机最好堆叠

keepalived+mysqlreplication

需要申请一个vip 内网

1 时间需要同步

2 开通nat

镜像构建

1. 服务特性

网络，磁盘，万兆，连通外网

2. 服务器网卡配置

管理端 IP 配置在万兆，能够连通外网

3. 操作系统配置

centos7 (webserver)

4. raid 配置和分区设置

A1-3 [CPU:24C (E5-2620V3*2)
内存:128G (16G*8) 硬盘:600G (1T)
系统 raid1) , 1.5T (数据 raid10)
标配实体机 (普通应用) ||26500]

5. 机架部署网络拓扑

集群内服务器需要放置在不同机柜，其他无要求。

1 时间需要同步

2 开通nat

kubernetes master 节点

1. 服务特性

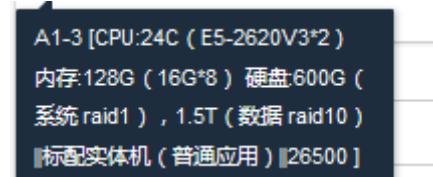
2. 服务器网卡配置

管理端 IP 配置在万兆网卡

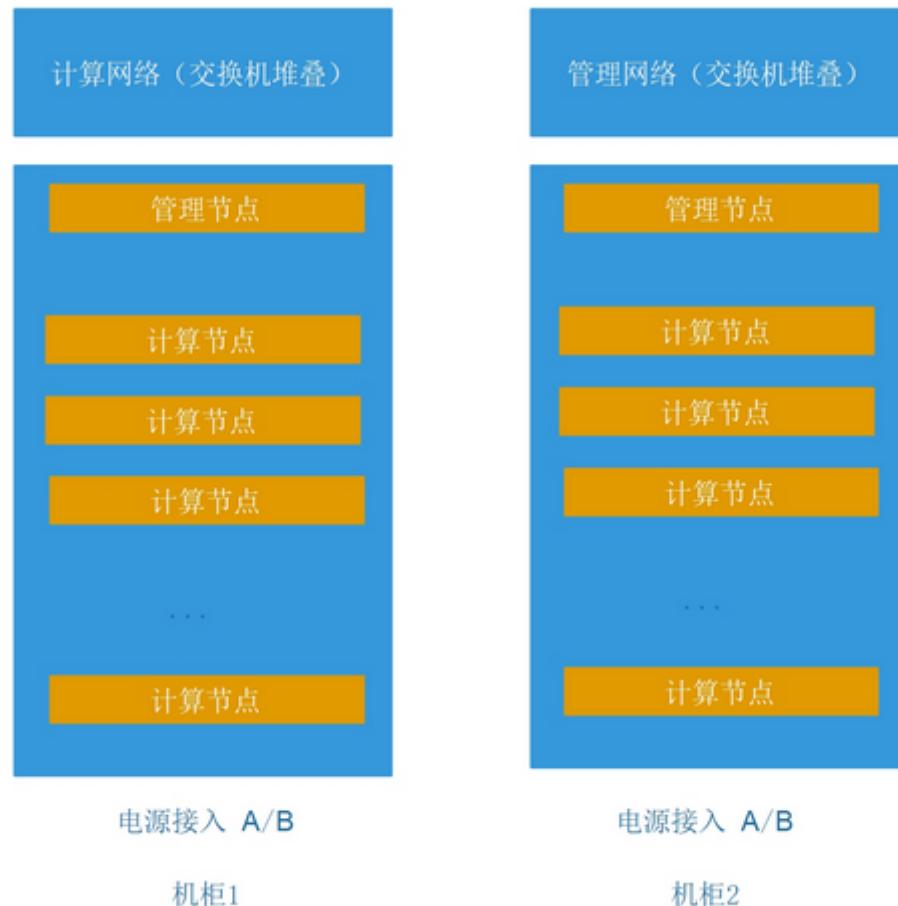
3. 操作系统配置

centos 7 (webserver)

4. raid 配置和分区设置



5. 机架部署网络拓扑



管理节点放置在两个机柜，且上联同一堆叠组的不同交换机下。

1 时间需要同步

2 开通nat

kubernetes node 计算节点

1. 服务特性

cpu, 内存, 网络, 万兆, 能连通外网, dns 配置问题

2. 服务器网卡配置

网卡配置	交换机接口模式	vlan	IP段	备注
br0 -> bond0 ->千兆 (eth0+eth1) mode 4	lacp+access			宿主机管理网段
万兆 eth2 eth3	lacp+trunk			容器网段

ovs bond: (下面的命令不需要操作)

ovs-vsctl add-br leenginebr0

ovs-vsctl add-bond leenginebr0 bond1 eth2 eth3 bond_mode=balance-tcp lacp=active

检测bond状态

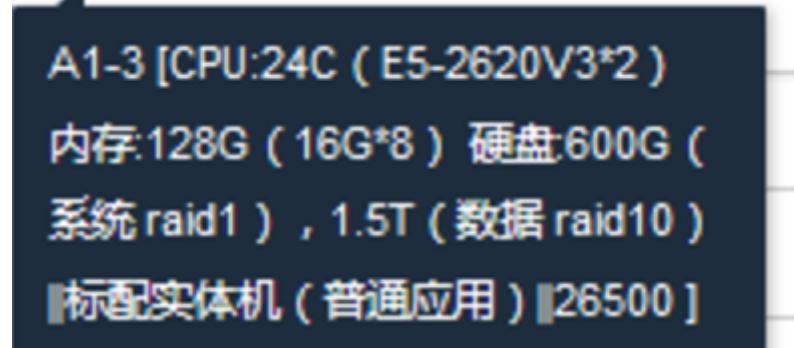
ovs-appctl bond/show bond1

ovs-appctl lacp/show bond1

3. 操作系统配置

CentOS Linux release 7.2.1511 (webserver)

4. raid 配置和分区设置



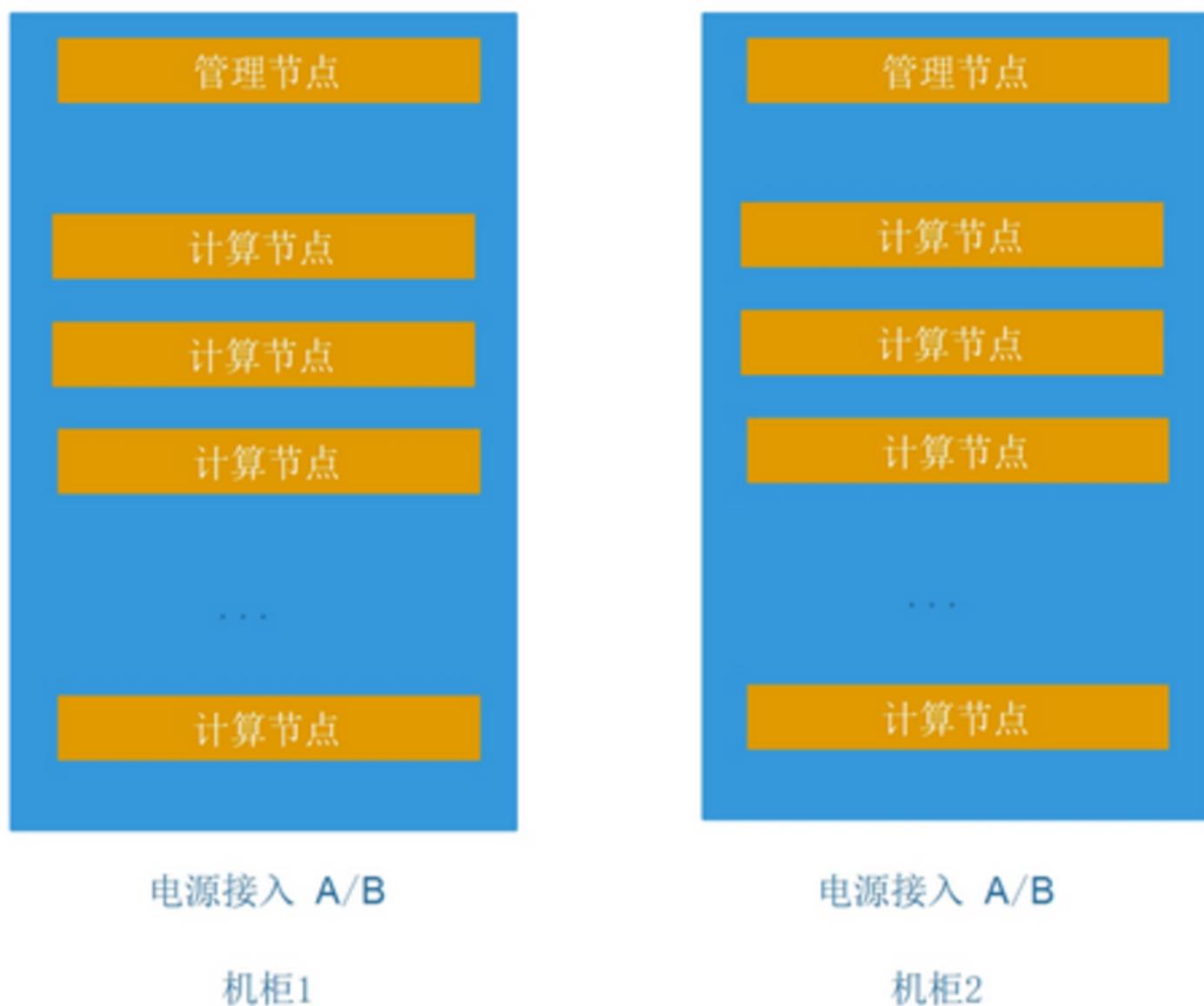
要存在/letv 和/data 分区

/data 分区作为数据盘，要最大

5. 机架部署网络拓扑

计算网络（交换机堆叠）

管理网络（交换机堆叠）



主机上联交换机必须做堆叠，主机管理网卡和虚机流量网卡都做冗余(bonding, mode4)。

为保障独立的故障域，计算节点最小上架规模为2个机柜，IP根据机柜分配。

同机柜计算节点 电源接入A/B路各一半。

计算节点bonding网卡上联同一堆叠组的不同交换机，提高故障冗余度。

服务器网卡上联，交换机接口配置统一为port channel lacp (trunk/access)模式。

1 时间需要同步

2 开通nat

网络基础及bond技术

OSI参考模型：

1.物理层 (Physical Layer)

OSI模型的最低层或第一层，为上层协议提供了一个传输数据的物理媒体。

在这一层，协议数据单元为比特 (bit) 。

在物理层的互联设备包括：集线器 (Hub) 、中继器 (Repeater) 等。

2.数据链路层 (Datalink Layer)

OSI模型的第二层，它控制网络层与物理层之间的通信，其主要功能是在不可靠的物理介质上提供可靠的传输。

在这一层，协议数据单元为帧 (frame) 。

在数据链路层的互联设备包括：网桥 (Bridge) 、交换机 (Switch) 等。

3.网络层 (Network Layer)

OSI模型的第三层，其主要功能是将网络地址翻译成对应的物理地址，并决定如何将数据从发送方路由到接收方。

在这一层，协议数据单元为数据包 (packet) 。

网络层协议的代表包括：互联网协议IP、地址解析协议ARP、反向地址转换协议RARP、以太网协议Ethernet等。

在网络层的互联设备包括：路由器 (Router) 等。

4.传输层 (Transport Layer)

OSI模型中最重要的一层，是第一个主机到主机的层次。其主要功能是负责将上层数据分段并提供端到端的、可靠的或不可靠的传输。

在这一层，协议数据单元为数据段 (segment) 。

传输层协议的代表包括：TCP、UDP、SPX等。

5.会话层 (Session Layer)

OSI模型的第五层

6.表示层 (Presentation Layer)

OSI模型的第六层。表示层的数据转换包括数据的解密和加密、压缩、格式转换等。

7.应用层 (Application Layer)

OSI模型的第七层，应用层提供的服务包括文件传输、文件管理以及电子邮件的信息处理。

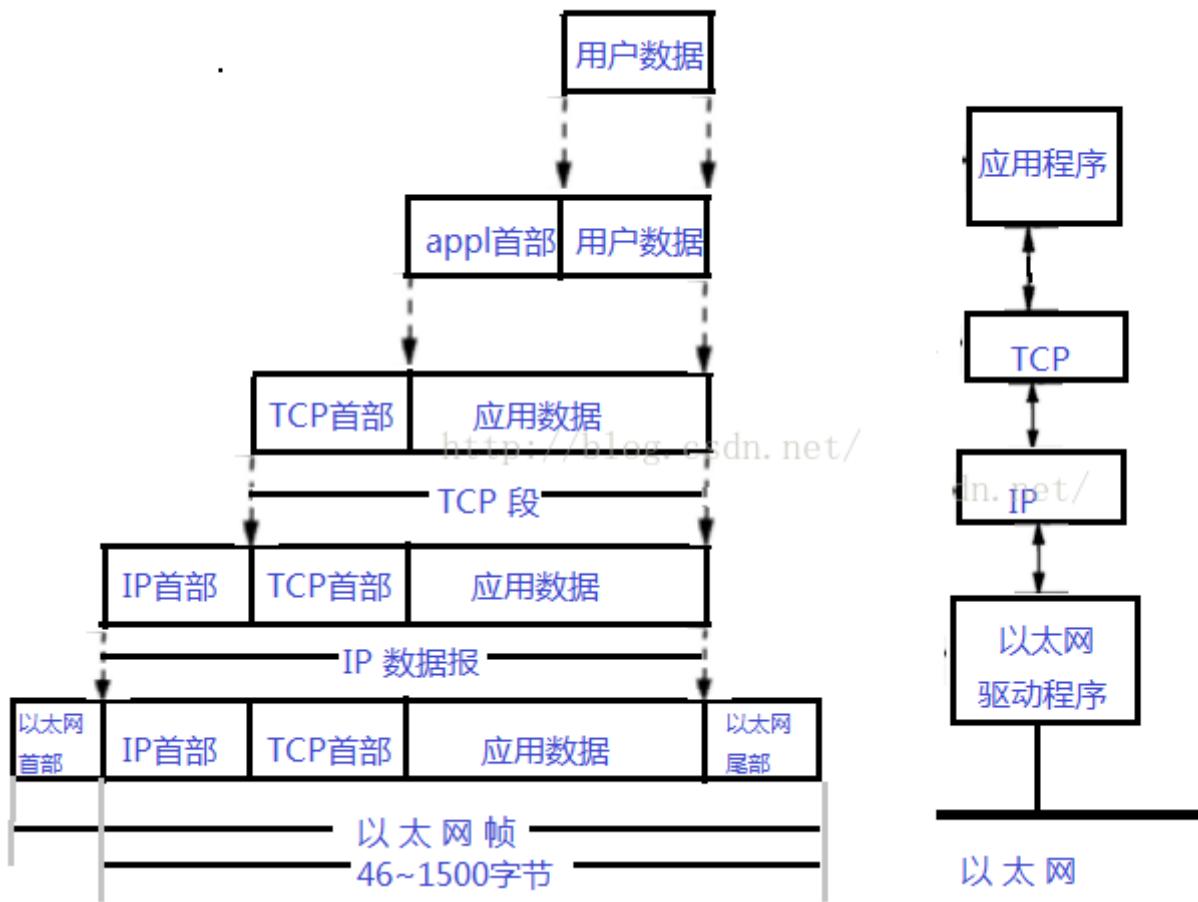
应用层协议的代表包括：FTP、Telnet、SMTP、HTTP、POP3、HTTPS等。

在应用层的互联设备包括：网关 (Gateway) 等。

TCP/IP协议栈

TCP/IP网络协议栈分为应用层（对应OSI的上面3层）、传输层、网络层和链路层（对应OSI的物理层+链路层）四层。

TCP/IP数据包的封包：应用层将数据通过TCP/IP协议栈逐层向下传递，其下的每层接到来自上层的数据时，根据每层的协议都要在其数据的前端添加首部信息进行封装。不同的协议层对数据包有不同的称谓，在传输层叫做段 (segment)，在网络层叫做数据报(datagram)，在链路层叫做帧(frame)。在经过链路层时，数据封装成帧后发给物理层的传输介质上，到达目标主机后，每层协议再逐层剥掉其首部，最后递交给目标主机应用层，进行应用程序处理，数据与源主机发送的数据一致。



用户数据

1. 在物理层经TCP协议封装后，传到下一层，TCP为了保证不发生丢包，就在TCP头部给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。接收端实体对已成功收到的包发回一个相应的确认(ACK)。对于TCP协议，不存在分片的问题，因为TCP报头的选项字段有MSS字段，规定一个TCP包最大可传输的字节数，一般是 $1500 - 20 - 20 = 1460$ 字节。也就是说到达IP封包的时候最大可承载数据只有1460个字节，这样就可以避免分片的问题。但是对于UDP这一类的协议，分片操作还是交给IP完成，虽然这样可以增加负载的效率，但是稳定性会受到很大的影响。
2. 又经IP协议封装头部，头部中有字段记录是否分片（如果上层传来的是不分片，但数据实际长度大于1500-IP协议头部，则数据包被丢弃，并将一个ICMP错误返回给源主机），IP数据包的长度为16，也就是说最大为65535个字节。但是我们知道数据链路层规定MTU最大传输单元为1500个字节。当IP数据包的大小超过了1500，IP数据包必须要进行分片处理。这还只是初始发送端的分片，在网络中如果某个路由器的MTU小于1500，那么还需要在路由器端做分片，这就给IP包的完整接收造成了很大的不确定性（我们知道数据包达到终点不一定是按照顺序的），因为万一某一个分片丢失，可能会造成整个IP重传，当然前提是高层协议支持重传。
3. 又经以太网帧协议封装，封装源MAC地址，目的MAC地址，报文类型为IP数据包，数据最大1500传输单元。使用ifconfig可以看到MTU值为1500。在封装以太网数据包时，根据目的IP地址，查找目的MAC地址，每台主机都维护一个ARP缓存表，可以用arp -a命令查看。缓存表中的表项有过期时间(一般为20分钟)，如果20分钟内没有再次使用某个表项，则该表项失效，如果查不到本地有这个IP对应的MAC地址，则会封装一个报文类型为ARP类型的以太网帧格式，数据内容为ARP请求数据，然后广播给局域网所有机器，只是一个二层的局域网内，不能跨三层路由，当所有机器收到这个报文后，解开，发现目标IP地址不是自己的，抛出去，只有目标机器才会返回一个类型为ARP的以太网帧报文，数据内容为ARP应答。现在活得目标主机的MAC地址，交换机会保存哪个端口对应的哪个MAC地址。下次要有请求获取ARP，交换机就可以直接返回。源主机获取到该以太网帧以后，在本地ARP表保存这条IP到MAC映射，同时修改装有IP数据包的以太网帧，写进去目的MAC地址，然后发送出去。

RARP协议是通过目的物理MAC地址，活得对应的IP地址。

虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP和RARP属于链路层，IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报，但是从功能上划分，ICMP、IGMP与IP同属于网络层，TCP和UDP属于传输层。

TCP有6个标志位：URG、ACK、PSH、RST、SYN和FIN。

SYN表示建立连接，FIN表示关闭连接；ACK表示响应；PSH表示有 DATA数据传输；RST表示连接重置；URG表示紧急指针字段有效。

三次握手可概括为：

请求连接，收到应答，再次请求。 SYN ----> SYN -----> ACK

三次握手是基于请求应答模式的，防止已过期的连接再次传到被连接的主机，三次握手改成仅需要两次握手，死锁是可能发生，从而解决了丢包问题。

四次挥手可概括为：

主动请求释放，主端收到应答，对端请求释放，对端收到应答。 FIN----->ACK----->FIN----->ACK

四次挥手确保，连接释放，确保数据能够完成传输。挥手需要双发都同意才能结束。

网卡Bond模式

前网卡绑定mode共有七种(0~6)bond0、bond1、bond2、bond3、bond4、bond5、bond6

常用的有三种：

mode=0：平衡负载模式，有自动备援，但需要交换机支援及设定。

mode=1：自动备援模式，其中一条线若断线，其他线路将会自动备援。

mode=6：平衡负载模式，有自动备援，不必交换机支援及设定。

磁盘阵列RAID

常用磁盘类型

常见的磁盘接口类型有IDE、SATA、SCSI、SAS、FC、SSD。

- IDE是俗称的并口，SATA是俗称的串口，这两种硬盘是个人电脑和低端服务器常见的硬盘，是普通PC的标准接口。
- SCSI是广泛应用于小型机上的高速数据传输技术。
- SAS就是串口的SCSI接口。一般服务器硬盘采用这两类接口，其性能比上述两种硬盘要高，稳定性更强，但是价格高，容量小，噪音大。
- FC是光纤通道，和SCIS接口一样，光纤通道最初也不是为硬盘设计开发的接口技术，是专门为网络系统设计的，但随着存储系统对速度的需求，才逐渐应用到硬盘系统中。
- SSD也称作电子硬盘或者固态电子盘，是由控制单元和固态存储单元（DRAM或FLASH芯片）组成的硬盘。固态硬盘的接口规范和定义、功能及使用方法上与普通硬盘的相同，在产品外形和尺寸上也与普通硬盘一致。新一代的固态硬盘普遍采用SATA-2接口。但其成本较高。

RAIDx

RAID称为廉价磁盘冗余阵列。RAID的基本原理是把多个便宜的小磁盘组合到一起，成为一个磁盘组，使性能达到或超过一个容量巨大、价格昂贵的磁盘。

目前 RAID技术大致分为两种：基于硬件的RAID技术和基于软件的RAID技术。其中在Linux下通过自带的软件就能实现RAID功能，Windows也有软件。这样就可以实现将几个物理磁盘合并成一个更大的虚拟设备，从而达到性能改进和数据冗余的目的。当然基于硬件的RAID解决方案比基于软件RAID技术在使用性能和服务性能上稍胜一筹，具体表现在检测和修复多位错误的能力、错误磁盘自动检测和阵列重建等方面。

RAIDx介绍：

RAID最常用的RAID0、RAID1、RAID5、RAID6、RAID10。

- RAID 0是两盘一起读写，如果一个坏了那么数据全丢失，必须要有两个以上硬盘驱动器；
- RAID 1是一块写，一块用来备份，坏一块无所谓，RAID 1可以用于两个或 2^*N 个磁盘，使用0块或更多的备用磁盘，每次写数据时会同时写入镜像盘。这种阵列可靠性很高，但其有效容量减小到总容量的一半，同时这些磁盘的大小应该相等，否则总容量只具有最小磁盘的大小。RAID 1技术支持“热替换”，即不断电的情况下对故障磁盘进行更换，更换完毕只要从镜像盘上恢复数据即可；
- RAID 0+1: 也被称为RAID 10标准，实际是将RAID 0和RAID 1标准结合的产物，在连续地以位或字节为单位分割数据并且并行读/写多个磁盘的同时，为每一块磁盘作磁盘镜像进行冗余。它的优点是同时拥有RAID 0的超凡速度和RAID 1的数据高可靠性，但是CPU占用率同样也更高，而且磁盘的利用率比较低；
- RAID 2很少用，将数据条块化地分布于不同的硬盘上，条块单位为位或字节，并使用称为“加重平均纠错码（海明码）”的编码技术来提供错误检查及恢复；
- RAID 3也很少用，同RAID 2非常类似，都是将数据条块化分布于不同的硬盘上，区别在于RAID 3使用简单的奇偶校验，并用单块磁盘存放奇偶校验信息；
- RAID 4也很少用，同样也将数据条块化并分布于不同的磁盘上，但条块单位为块或记录。RAID 4使用一块磁盘作为奇偶校验盘，每次写操作都需要访问奇偶盘，这时奇偶校验盘会成为写操作的瓶颈；
- RAID 5不单独指定奇偶盘，而是在所有磁盘上交叉地存取数据及奇偶校验信息。在RAID 5上，读/写指针可同时对阵列设备进行操作，提供了更高的数据流量。RAID 5更适合于小数据块和随机读写的数据。可以用在三块或更多的磁盘上，并使用0块或更多的备用磁盘。允许坏一块盘，但是最少需要三块盘来做。
- RAID 6增加了第二个独立的奇偶校验信息块。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高，即使两块磁盘同时失效也不会影响数据的使用。但RAID 6需要分配给奇偶校验信息更大的磁盘空间，相对于RAID 5有更大的“写损失”，因此“写性能”非常差。较差的性能和复杂的实施方式使得RAID 6很少得到实际应用。允许坏2块盘，但最少需要四块盘。
- RAID 7是一种新的RAID标准，其自身带有智能化实时操作系统和用于存储管理的软件工具，可完全独立于主机运行，不占用主机CPU资源。RAID 7可以看作是一种存储计算机（Storage Computer），它与其他RAID标准有明显区别。
- 可以如RAID 0+1结合多种RAID规范来构筑所需的RAID阵列，RAID 5+3（RAID 53）也是一种应用较为广泛的阵列形式。

一般常用的RAID阶层，分别是RAID 0、RAID1、RAID 2、RAID 3、RAID 4以及RAID 5，再加上二合一型 RAID 0+1〔或称RAID 10〕。RAID级别的优、缺点：

RAID级别	相对优点	相对缺点
RAID 0	存取速度最快	没有容错
RAID 1	完全容错	成本高
RAID 2	带海明码校验	数据冗余多，速度慢
RAID 3	写入性能最好	没有多任务功能

RAID 4	具备多任务及容错功能	磁盘驱动器造成性能瓶颈
RAID 5	具备多任务及容错功能	写入时有overhead
RAID 0+1/RAID 10	速度快、完全容错	成本高

LVM

MBR与GPT区别：

- MBR方式，开机管理程序纪录区与分区表通通放在磁盘的第一个扇区，这个扇区通常是 512Bytes 的大小。
第一个扇区 512Bytes 会有这两个数据：
 - 主要开机记录区(Master Boot Record, MBR):可以安装开机管理程序的地方，有446 Bytes
 - 分区表(partition table):记录整颗硬盘分区的状态，有64 Bytes

由于分区表所在区块仅有64 Bytes容量，因此最多仅能有四组记录区，每组记录区记录了该区段的启始与结束的柱面号码。

磁盘的 MBR 分区方式中，主要与扩展分区最多可以有四个，如果磁盘容量大于 2TB 以上时，系统会自动使用 GPT 分区方式来处理磁盘分区。GPT 分区已经没有延伸与逻辑分区的概念，可以想像成所有的分区都是主分区！

硬盘默认的分区表仅能写入四组分区信息，这四组分区信息我们称为主要(Primary)或延伸(Extended)分区，主要分区与延伸分区最多可以有四笔(硬盘的限制)，延伸分区最多只能有一个(操作系统的限制)，逻辑分区是由延伸分区持续切割出来的分区，能够被格式化后，作为数据存取的分区为主要分区与逻辑分区。延伸分区无法格式化，逻辑分区的数量依操作系统而不同，在Linux系统中SATA硬盘已经可以突破63个以上的分区限制；

一个硬盘可以有1到3个主分区和1个扩展分区,也可以只有主分区而没有扩展分区,但主分区必须至少有1个,扩展分区则最多只有1个,且主分区+扩展分区总共不能超过4个。逻辑分区可以有若干个。

1. 查看当前所有的磁盘信息 fdisk -l

2. 查看分区挂在目录 df -lh

3. 选择磁盘创建分区(分区又分主分区，扩展分区、逻辑分区，扩展分区不占大小，在扩展分区上创建逻辑分区
TODO) fdisk /dev/sdb

4. 格式化分区 mkfs.ext4 /dev/sdb2

5. 挂载分区到目录上 mount /dev/sdb1 /my_mount1

6. 制定分区格式 fdisk -l根据向导中提示修改 LVM格式为8e

7. 卸载分区 umount /dev/sdb1

8. 删除分区 fdisk -l根据向导删除

LVM是逻辑卷管理的简写，它将一个或多个硬盘的分区在逻辑上集合，相当于一个大硬盘来使用，当硬盘的空间不够使用的时候，可以继续将其它的硬盘的分区加入其中，这样可以实现磁盘空间的动态管理，相对于普通的磁盘分区有很大的灵活性。

LVM提供了一个抽象的盘卷，在盘卷上建立文件系统。首先我们讨论以下几个LVM术语：

物理存储介质 (The physical media) : 这里指系统的存储设备：硬盘。

- PV: 是物理的磁盘分区
- VG: LVM中的物理的磁盘分区，也就是PV，必须加入VG，可以将VG理解为一个仓库或者是几个大的硬盘。
- LV: 也就是从VG中划分的逻辑分区

要创建一个LVM系统，一般需要经过以下步骤：

1. 创建分区 fdisk -l, 一定指定分区格式为8e

2. 创建PV pvcreate 分区名 查看已经存在的PV: pvdisplay

3. 创建VG `vgcreate VG名称 PV名` 可利用已经存在的VG，同一个VG下的一组PV构成一个VG。查看VG：
`vgdisplay`
4. 从VG中划分一个LV `lvcreate -L 100M -n lv名称 PV名` 显示所有LV信息：`lvdisplay` 将LV主设备、此设备号写到`/sys/fs/cgroup/`，限制分区IOPS。
5. LV格式化文件系统 `mkfs -t ext3 /dev/PV名/LV名`
6. 挂载LV到一个目录 `mount /dev/PV名/LV名 /root/test目录`

五 In Project

Matrix+Beehive+Mcluster

要立即开

现有部署情况，现在支持最大情况

Leengine

要立即开

现有部署情况，现在支持最大情况

Marsone

要立即开

Kubernetes持久化

要立即开

一片天

要立即开

大型项目文章篇

要立即开