# JAVA基础

Object方法

(1)public Object();

默认构造方法

(2)private static native void registerNatives();

将C/C++中的方法映射到Java中的native方法，实现方法命名的解耦。

(3)protected native Object clone() throws CloneNotSupportedException;

创建并返回此对象的一个副本。此对象与原对象分别占用不同的堆空间。

(4)public boolean equals(Object obj);

指示某个其他对象是否与此对象“相等”。重写equals()方法必须重写hasCode()方法。

(5)hashCode()

返回该对象的哈希码值。

(6)public String toString();

返回该对象的字符串表示。return getClass().getName() + "@" + Integer.toHexString(hashCode());

(7)public final native Class<?> getClass();

返回一个对象的运行时类。类是是对具有一组相同特征或行为的实例的抽象并进行描述，对象则是此类所描述的特征或行为的具体实例。

(8)protected void finalize() throws Throwable { }

当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

(9)wait()

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

(10)wait(long timeout)

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量。

(11)wait(long timeout， int nanos)

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

(12)notify()

唤醒在此对象监视器上等待的单个线程。

(13)notifyAll()

唤醒在此对象监视器上等待的所有线程。

#### Java语言的三大特性即是：封装、继承、多态

（1）封装：

属性可用来描述同一类事物的特征，方法可描述一类事物可做的操作。封装就是把属于同一类事物的共性（包括属性与方法）归到一个类中，以方便使用。

1。 概念：封装也称为信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部， 尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。系统的其他部分只有通过包裹在数据外面的被授权的操作来与这个抽象数据类型交流与交互。 也就是说，用户无需知道对象内部方法的实现细节，但可以根据对象提供的外部接口(对象名和参数)访问该对象。

2。好处：(1)实现了专业的分工。 将能实现某一特定功能的代码封装成一个独立的实体后，各程序员可以在需要的时候调用，从而实现了专业的分工。(2)隐藏信息，实现细节。通过控制访问权限 可以将可以将不想让客户端程序员看到的信息隐藏起来，如某客户的银行的密码需要保密，只能对该客户开发权限。

（2）继承：

个性对共性的属性与方法的接受，并加入个性特有的属性与方法，目的：实现代码的复用。

1。概念：一个类继承另一个类，则称继承的类为子类，被继承的类为父类。

2。理解：子类与父类的关系并不是日常生活中的父子关系，子类与父类而是一种特殊化与一般化的关系，是is-a的关系，子类是父类更加详细的分类。如 class dog extends animal，就可以理解为dog is a animal。注意设计继承的时候，若要让某个类能继承，父类需适当开放访问权限，遵循里氏代换原则，即向修改关闭对扩展开放，也就是开-闭原则。

4。结果：继承后子类自动拥有了父类的属性和方法，但特别注意的是，父类的私有属性和构造方法并不能被继承。

另外子类可以写自己特有的属性和方法，目的是实现功能的扩展，子类也可以复写父类的方法即方法的重写。

（3）多态：

多态是以封装和继承为基础的。多态就是在抽象的层面上实施一个统一的行为，到个体（具体）的层面上时，这个统一的行为会因为个体（具体）的形态特征而实施自己的特征行为。（针对一个抽象的事，对于内部个体又能找到其自身的行为去执行。）

1.概念：相同的事物，调用其相同的方法，参数也相同时，但表现的行为却不同。

2.理解：Father f=new Son()子类以父类的身份出现，但做事情时还是以自己的方法实现，需要向上转型(upcast)，由JVM自动实现的是安全的，但向下转型(downcast)是不安全的，需要强制转换。子类以父类的身份出现时自己特有的属性和方法将不能使用。

多态：后期绑定实现多态，Java中有两种形式可以实现多态：继承和接口。

方法的重写Overriding和重载Overloading是Java多态性的不同表现；

Java实现多态有三个必要条件：继承、重写、向上转型。

Father f=new Son()

向上转型：

1。只能访问父类中拥有的方法和属性，而对于子类中存在而父类中不存在的方法，该引用是不能使用的，尽管是重载该方法。

2。若子类重写了父类中的某些方法，在调用该些方法的时候，必定是使用子类中定义的这些方法（动态连接、动态调用）。

#### 静态绑定和动态绑定

重载(Overload)的方法使用静态绑定完成，而重写(Override)的方法则使用动态绑定完成。

静态绑定发生在编译时期，动态绑定发生在运行时

使用private或static或final修饰的变量或者方法，使用静态绑定。而虚方法（可以被子类重写的方法）则会根据运行时的对象进行动态绑定。

静态绑定使用类信息来完成，而动态绑定则需要使用对象信息来完成。

#### 重载和重写的区别

　首先我们来讲讲：**重载(Overloading)**

1. 方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。

重载Overloading是一个类中多态性的一种表现。

（2） [Java](http://java.chinaitlab.com/)的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。

调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法， 这就是多态性。

（3） 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

然后我们再来谈谈 **重写（Overriding）**

（1） 父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。在[Java](http://java.chinaitlab.com/)中，子类可继承父类中的方法，而不需要重新编写相同的方法。

但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。

方法重写又称方法覆盖。

（2）若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。

如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。

（3）子类函数的访问修饰权限不能少于父类的；

概念：即调用对象方法的机制。

**动态绑定**的内幕：

 1、编译器检查对象声明的类型和方法名，从而获取所有候选方法。试着把上例Base类的test注释掉，这时再编译就无法通过。

 2、重载决策：编译器检查方法调用的参数类型，从上述候选方法选出唯一的那一个（其间会有隐含类型转化）。

如果编译器找到多于一个或者没找到，此时编译器就会报错。试着把上例Base类的test(byte b)注释掉，这时运行结果是1 1。

 3、若方法类型为priavte static final ，java采用静态编译，编译器会准确知道该调用哪个方法。

 4、当程序运行并且使用动态绑定来调用一个方法时，那么虚拟机必须调用对象的实际类型相匹配的方法版本。

在例子中，b所指向的实际类型是TestOverriding，所以b.test(0)调用子类的test。

但是，子类并没有重写test(byte b)，所以b.test((byte)0)调用的是父类的test(byte b)。

如果把父类的(byte b)注释掉，则通过第二步隐含类型转化为int，最终调用的是子类的test(int i)。

**学习总结**：

    多态性是面向对象编程的一种特性，和方法无关，  
    简单说，就是同样的一个方法能够根据输入数据的不同，做出不同的处理，即方法的  
 重载——有不同的参数列表（静态多态性）而当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法，即在子类中重写该方法——相同参数，不同实现（动态多态性）

**重写方法的规则**：

1、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

2、返回的类型必须与被重写的方法的返回类型相同，否则不能称其为重写而是重载。

3、访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）

4、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：

父类的一个方法申明了一个检查异常IOException，在重写这个方法是就不能抛出Exception，只能抛出IOException的子类异常，可以抛出非检查异常。

**而重载的规则：**

1、必须具有不同的参数列表；

2、可以有不同的返回类型，只要参数列表不同就可以了；

3、可以有不同的访问修饰符；

4、可以抛出不同的异常；

**重写与重载的区别在于：**

重写多态性起作用，对调用被重载过的方法可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值。

用好重写和重载可以设计一个结构清晰而简洁的类，可以说重写和重载在编写代码过程中的作用非同一般。

#### [多态性实现机制——静态分派与动态分派](http://blog.csdn.net/ns_code/article/details/17965867)

##### 方法解析

    Class文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在Class文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址。这个特性给[**Java**](http://lib.csdn.net/base/java)带来了更强大的动态扩展能力，使得可以在类运行期间才能确定某些目标方法的直接引用，称为动态连接，也有一部分方法的符号引用在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。这在前面的“Java内存区域与内存溢出”一文中有提到。

    静态解析成立的前提是：方法在程序真正执行前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在编译器进行编译时就必须确定下来，这类方法的调用称为解析。

    在Java语言中，符合“编译器可知，运行期不可变”这个要求的方法主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法都不可能通过继承或别的方式重写出其他的版本，因此它们都适合在类加载阶段进行解析。

   Java虚拟机里共提供了四条方法调用字节指令，分别是：

* invokestatic：调用静态方法。
* invokespecial：调用实例构造器<init>方法、私有方法和父类方法。
* invokevirtual：调用所有的虚方法。
* invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

    只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器和父类方法四类，它们在类加载时就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法（还包括final方法），与之相反，其他方法就称为虚方法（final方法除外）。这里要特别说明下final方法，虽然调用final方法使用的是invokevirtual指令，但是由于它无法覆盖，没有其他版本，所以也无需对方发接收者进行多态选择。Java语言规范中明确说明了final方法是一种非虚方法。

    解析调用一定是个静态过程，在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用转化为可确定的直接引用，不会延迟到运行期再去完成。而分派调用则可能是静态的也可能是动态的，根据分派依据的宗量数（方法的调用者和方法的参数统称为方法的宗量）又可分为单分派和多分派。两类分派方式两两组合便构成了静态单分派、静态多分派、动态单分派、动态多分派四种分派情况。

##### 静态分派

    所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派，静态分派的最典型应用就是多态性中的方法重载。静态分派发生在编译阶段，因此确定静态分配的动作实际上不是由虚拟机来执行的。下面通过一段方法重载的示例程序来更清晰地说明这种分派机制：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/17965867) [copy](http://blog.csdn.net/ns_code/article/details/17965867) [print?](http://blog.csdn.net/ns_code/article/details/17965867)

1. class Human{
2. }
3. class Man extends Human{
4. }
5. class Woman extends Human{
6. }
8. public class StaticPai{
10. public void say(Human hum){
11. System.out.println("I am human");
12. }
13. public void say(Man hum){
14. System.out.println("I am man");
15. }
16. public void say(Woman hum){
17. System.out.println("I am woman");
18. }
20. public static void main(String[] args){
21. Human man = new Man();
22. Human woman = new Woman();
23. StaticPai sp = new StaticPai();
24. sp.say(man);
25. sp.say(woman);
26. }
27. }

class Human{

}

class Man extends Human{

}

class Woman extends Human{

}

public class StaticPai{

public void say(Human hum){

System.out.println("I am human");

}

public void say(Man hum){

System.out.println("I am man");

}

public void say(Woman hum){

System.out.println("I am woman");

}

public static void main(String[] args){

Human man = new Man();

Human woman = new Woman();

StaticPai sp = new StaticPai();

sp.say(man);

sp.say(woman);

}

}

    上面代码的执行结果如下：

    I am human  
    I am human

   以上结果的得出应该不难分析。在分析为什么会选择参数类型为Human的重载方法去执行之前，先看如下代码：

Human man = new Man（）;

    我们把上面代码中的“Human”称为变量的静态类型，后面的“Man”称为变量的实际类型。静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的，而实际类型变化的结果在运行期才可确定。

    回到上面的代码分析中，在调用say（）方法时，方法的调用者（回忆上面关于宗量的定义，方法的调用者属于宗量）都为sp的前提下，使用哪个重载版本，完全取决于传入参数的数量和数据类型（方法的参数也是数据宗量）。代码中刻意定义了两个静态类型相同、实际类型不同的变量，可见**编译器（不是虚拟机，因为如果是根据静态类型做出的判断，那么在编译期就确定了）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。**并且静态类型是编译期可知的，所以在编译阶段，Javac编译器就根据参数的静态类型决定使用哪个重载版本。这就是静态分派最典型的应用。

##### 动态分派

   动态分派与多态性的另一个重要体现——方法覆写有着很紧密的关系。向上转型后调用子类覆写的方法便是一个很好地说明动态分派的例子。这种情况很常见，因此这里不再用示例程序进行分析。很显然，在判断执行父类中的方法还是子类中覆盖的方法时，如果用静态类型来判断，那么无论怎么进行向上转型，都只会调用父类中的方法，但实际情况是，根据对父类实例化的子类的不同，调用的是不同子类中覆写的方法，很明显，这里是要根据变量的实际类型来分派方法的执行版本的。而实际类型的确定需要在程序运行时才能确定下来，这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

##### 单分派和多分派

前面给出：方法的接受者（亦即方法的调用者）与方法的参数统称为方法的宗量。但分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。

    为了方便理解，下面给出一段示例代码：

1. class Eat{
2. }
3. class Drink{
4. }
6. class Father{
7. public void doSomething(Eat arg){
8. System.out.println("爸爸在吃饭");
9. }
10. public void doSomething(Drink arg){
11. System.out.println("爸爸在喝水");
12. }
13. }
15. class Child extends Father{
16. public void doSomething(Eat arg){
17. System.out.println("儿子在吃饭");
18. }
19. public void doSomething(Drink arg){
20. System.out.println("儿子在喝水");
21. }
22. }
24. public class SingleDoublePai{
25. public static void main(String[] args){
26. Father father = new Father();
27. Father child = new Child();
28. father.doSomething(new Eat());
29. child.doSomething(new Drink());
30. }
31. }

class Eat{

}

class Drink{

}

class Father{

public void doSomething(Eat arg){

System.out.println("爸爸在吃饭");

}

public void doSomething(Drink arg){

System.out.println("爸爸在喝水");

}

}

class Child extends Father{

public void doSomething(Eat arg){

System.out.println("儿子在吃饭");

}

public void doSomething(Drink arg){

System.out.println("儿子在喝水");

}

}

public class SingleDoublePai{

public static void main(String[] args){

Father father = new Father();

Father child = new Child();

father.doSomething(new Eat());

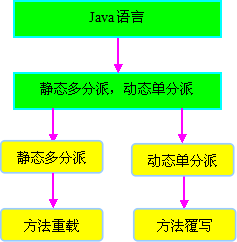
child.doSomething(new Drink());

}

}

    运行结果应该很容易预测到，如下：

    爸爸在吃饭  
    儿子在喝水  
  我们首先来看编译阶段编译器的选择过程，即静态分派过程。这时候选择目标方法的依据有两点：一是方法的接受者（即调用者）的静态类型是Father还是Child，二是方法参数类型是Eat还是Drink。因为是根据两个宗量进行选择，所以Java语言的静态分派属于多分派类型。  
  再来看运行阶段虚拟机的选择，即动态分派过程。由于编译期已经了确定了目标方法的参数类型（编译期根据参数的静态类型进行静态分派），因此唯一可以影响到虚拟机选择的因素只有此方法的接受者的实际类型是Father还是Child。因为只有一个宗量作为选择依据，所以Java语言的动态分派属于单分派类型。



    根据以上论证，我们可以总结如下：目前的Java语言（JDK1.6）是一门静态多分派、动态单分派的语言。

修饰词   本类    同包类 继承类 其他类

private√×××

无（默认）√√××

protected√√√×

public√√√√

#### 泛型:是类型擦除的

List<T>表示集合中元素都为T类型，具体类型在运行期决定，无需转型。可进行读写操作，应优先使用该泛型方式。

List<?>表示任意类型，是只读的(主动转型为Object)，多用于泛型方法的返回值。它不能增加修改，因为编译器无法校验类型安全，只能删除

List<Object>表示集合中元素都为Object类型，需转型。写入要向上转型，读取要向下转型

数据类型：

1）：基本数据类型：byte、short、int、long、float、double、char、boolean

2）：引用数据类型：数组、类、接口。

级别从低到高为：byte，char，short(这三个平级)-->int-->float-->long-->double

自动类型转换：从低级别到高级别，系统自动转的；

强制类型转换：什么情况下使用?把一个高级别的数赋给一个别该数的级别低的变量；

#### 抽象类与接口：抽象类和接口的共性：都是不断向上抽取的结果。

  抽象类：一般用于描述一个体系单元，将一组共性内容进行抽取特点：可以在类中定义抽象内容让子类实现，可以定义非抽象内容让子类直接使用。抽象类意义三句话概括：为其他子类提供一个公共的类型；封装子类中重复定义的内容；定义抽象方法，子类虽然有不同的实现，但是定义时一致的。

 接口：一般用于定义对象的扩展功能，是在继承之外还需这个对象具备的一些功能。接口的意义用三个词就可以概括:规范，扩展，回调。

抽象类和接口的区别：

1：抽象类只能被继承，而且只能单继承。接口需要被实现，而且可以多实现。

2：抽象类中可以定义非抽象方法，子类可以直接继承使用。接口中都是抽象方法，需要子类去实现。

3：抽象类使用的是  is a 关系。接口使用的 like a 关系。

4：抽象类的成员修饰符可以自定义。接口中的成员修饰符是固定的。全都是public的。

#### 创建对象的4种方式：

1采用new；2通过反射；3采用clone；4通过序列化机制

前2者都需要显式地调用构造方法。 造成耦合性最高的恰好是第一种，因此你发现无论什么框架，只要涉及到解耦必先减少new的使用。

通过clone方法生成的对象，不会再执行构造函数，只是在内存中进行数据拷贝

#### java四种引用<强软弱虚>：不同的引用类型主要体现在GC

强引用-gc不收，软引用-gc内存不够收掉，

弱引用-gc发现就收掉，虚引用。

反射Reflect

反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

反射机制提供功能：

在运行时判断任意一个对象所属的类；

在运行时构造任意一个类的对象；

在运行时判断任意一个类所具有的成员变量和方法；

在运行时调用任意一个对象的方法；

生成动态代理。

#### 抽象关键字abstract和哪些不可以共存？final ，private ， static

static表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态static代码块，但是Java语言中没有全局变量的概念。

final有“这是无法改变的”或者“终态的”含义，它可以修饰非抽象类、非抽象类成员方法和变量。

final类不能被继承，没有子类，final类中的方法默认是final的。final方法不能被子类的方法覆盖，但可以被继承。final成员变量表示常量，只能被赋值一次，赋值后值不再改变。final不能用于修饰构造方法。

注意：父类的private成员方法是不能被子类方法覆盖的，因此private类型的方法默认是final类型的。

java中final关键字有什么特点？

1、final关键字是一个修饰符，可以修饰类、方法和变量。

2、对于类：被final修饰的类是一个最终类，不可以被继承。

3、对于方法：被final修饰的方法是一个最终方法，不可以被覆盖。

4、对于变量：被final修饰的变量是一个常量，只能被赋值一次。

final优点：

final关键字提高了性能。JVM和Java应用都会缓存final变量。

final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。

使用final关键字，JVM会对方法、变量及类进行优化。

finally

异常处理时提供 finally 块来执行任何清除操作。如果抛出一个异常，那么相匹配的 catch 子句就会执行，然后控制就会进入 finally 块（如果有的话）。一般异常处理块需要。

finalize

Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作.finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。 Java中所有类都从Object类中继承finalize()方法。当垃圾回收器(garbage colector)决定回收某对象时，就会运行该对象的finalize()方法。

#### Java对象序列化Serializable

Java平台允许我们在内存中创建可复用的Java对象，但只有当JVM处于运行时对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java对象序列化就能够帮助我们实现该功能。

使用Java对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的"状态"，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

transient-序列化

transient是类型修饰符，只能用来修饰字段。在对象序列化的过程中，标记为transient的变量不会被序列化。

#### 说说进程，线程，协程之间的区别

1。进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。

2。线程是进程的一个实体，是cpu调度和分派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

#### 守护线程吗?它和非守护线程有什么区别

程序运行完毕，jvm会等待非守护线程完成后关闭，但是jvm不会等待守护线程。守护线程最典型的例子就是GC线程

--------------------------------------------------------------

#### 生产者消费者模型的作用是什么?

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

#### JAVA8

(1)Lambda表达式和函数式接口

Lambda（也称为闭包），它允许我们将函数当成参数传递给某个方法，或者把代码本身当作数据处理。

Arrays。asList( "a"， "b"， "d" )。forEach( e -> System。out。println( e ) );

(2)扩展了接口的含义：默认方法和静态方法

区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会被接口的实现类继承或者覆写

(3)方法引用

方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambda表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。

final Car car = Car。create( Car::new );

(4)重复注解：@Repeatable注解定义重复注解

拓宽注解的应用场景：注解几乎可以使用在任何元素上：局部变量、接口类型、超类和接口实现类，甚至可以用在函数的异常定义上。

(5)更好的类型推断

final Value< String > value = new Value<>();

value。getOrDefault( "22"， Value。defaultValue() );

Java 8增加了很多新的工具类（date/time类），并扩展了现存的工具类，以支持现代的并发编程、函数式编程等

(1)Optional

Optional仅仅是一个容器：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查

Optional< String > fullName = Optional。ofNullable( null );

System。out。println( "Full Name is set? " + fullName。isPresent() );

(2)Streams

java。util。stream：函数式编程

(3)Date/Time API

LocalDate和LocalTime类，Duration类

(4)Nashorn JavaScript引擎

(5)Base64

(6)并行数组

(7)并发性

 JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）。这意味着不会再有java。lang。OutOfMemoryError: PermGen问题，也不再需要你进行调优及监控内存空间的使用。

总结：

    PermGen space的全称是Permanent Generation space，是指内存的永久保存区域，这块内存主要是被JVM存放Class和Meta信息的，Class在被Loader时就会被放到PermGen space中，它和存放类实例(Instance)的Heap区域不同，GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很多CLASS的话，就很可能出现PermGen space错误，这种错误常见在web服务器对JSP进行pre compile的时候。如果你的WEB APP下都用了大量的第三方jar， 其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。

        PermGen空间状况：这部分内存空间将全部移除。JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。

        Metaspace 容量：默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

        另外，对于僵死的类及类加载器的垃圾回收将在元数据使用达到“MaxMetaspaceSize”参数的设定值时进行。适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

#### String、stringbuffer、stringbuilder 联系、区别、源码

public final class String implements java.io.Serializable， Comparable<String>， CharSequence

public final class StringBuffer extends AbstractStringBuilder implements java.io.Serializable， CharSequence

public final class StringBuilder extends AbstractStringBuilder implements java.io.Serializable， CharSequence

抽象类AbstractStringBuilder说明

1、变量及构造方法

char[] value;

int count;

AbstractStringBuilder() {

}

AbstractStringBuilder(int capacity) {

  value = new char[capacity];

}

2、扩容：newCapacity = value.length \* 2 + 2;

public void ensureCapacity(int minimumCapacity) {

  if (minimumCapacity > 0)

    ensureCapacityInternal(minimumCapacity);

}

 private void ensureCapacityInternal(int minimumCapacity) {

  // overflow-conscious code

  if (minimumCapacity – value.length > 0)

    expandCapacity(minimumCapacity);

}

void expandCapacity(int minimumCapacity) {

  int newCapacity = value.length \* 2 + 2;

  if (newCapacity - minimumCapacity < 0)

    newCapacity = minimumCapacity;

  if (newCapacity < 0) {

    if (minimumCapacity < 0) // overflow

      throw new OutOfMemoryError();

    newCapacity = Integer.MAX\_VALUE;

  }

  value = Arrays。copyOf(value， newCapacity);

}

3、append()方法

public AbstractStringBuilder append(String str) {

    if (str == null)

      return appendNull();

    int len = str.length();

    ensureCapacityInternal(count + len);

    str.getChars(0， len， value， count);

    count += len;

    return this;

}

总结：

1.三个类都是final类型，内部存储上本质都是字符数组char[] value；但String的成员变量value是final类型

2.StringBuilder和StringBuffer都是可变字符串，前者线程不安全，后者线程安全。

3.StringBuilder和StringBuffer的大部分方法均调用父类AbstractStringBuilder的实现。其扩容机制首先是把容量变为原来容量的2倍加2。最大容量是Integer.MAX\_VALUE，也就是0x7fffffff。

4.StringBuilder和StringBuffer的默认容量都是16，最好预先估计好字符串的大小避免扩容带来的时间消耗。

# JVM

#### 类初始化

  类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的[Java](http://lib.csdn.net/base/java)程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

* 遇到new、getstatic、putstatic、invokestatic这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的Java代码场景是：使用new关键字实例化对象时、读取或设置一个类的静态字段（static）时（被static修饰又被final修饰的，已在编译期把结果放入常量池的静态字段除外）、以及调用一个类的静态方法时。
* 使用Java.lang.refect包的方法对类进行反射调用时，如果类还没有进行过初始化，则需要先触发其初始化。
* 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

    虚拟机规定只有这四种情况才会触发类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。下面举一些例子来说明被动引用。

    1、通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类

    2、常量在编译阶段会存入调用它的类的常量池中，本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化

    3、通过数组定义来引用类，不会触发类的初始化

    最后看一下接口的初始化过程与类初始化过程的不同。

    接口也有初始化过程，上面的代码中我们都是用静态语句块来输出初始化信息的，而在接口中不能使用“static{}”语句块，但编译器仍然会为接口生成<clinit>类构造器，用于初始化接口中定义的成员变量（实际上是static final修饰的全局常量）。

   二者在初始化时最主要的区别是：当一个类在初始化时，要求其父类全部已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量），才会初始化该父接口。这点也与类初始化的情况很不同，回过头来看第2个例子就知道，调用类中的static final常量时并不会 触发该类的初始化，但是调用接口中的static final常量时便会触发该接口的初始化。

OOM：

JVM内存管理分为：分配和释放

内存分配：new-堆中(新生代的Eden区)

内存释放：gc()

#### 类加载过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。它们开始的顺序如下图所示：



    其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是按顺序**开始**，而不是按顺序**进行**或**完成**，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

    这里简要说明下Java中的绑定：绑定指的是把一个方法的调用与方法所在的类(方法主体)关联起来，对java来说，绑定分为静态绑定和动态绑定：

* 静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定。java当中的方法只有final，static，private和构造方法是前期绑定的。
* 动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

    下面详细讲述类加载过程中每个阶段所做的工作。

##### 加载

    加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

    1、通过一个类的全限定名来获取其定义的二进制字节流。

    2、将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

    3、在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

    注意，这里第1条中的二进制字节流并不只是单纯地从Class文件中获取，比如它还可以从Jar包中获取、从网络中获取（最典型的应用便是Applet）、由其他文件生成（JSP应用）等。

    相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

    加载阶段完成后，虚拟机外部的 二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建一个java.lang.Class类的对象，这样便可以通过该对象访问方法区中的这些数据。

    说到加载，不得不提到类加载器，下面就具体讲述下类加载器。

    类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类的加载阶段。对于任意一个类，都需要由它的类加载器和这个类本身一同确定其在就Java虚拟机中的唯一性，也就是说，即使两个类来源于同一个Class文件，只要加载它们的类加载器不同，那这两个类就必定不相等。这里的“相等”包括了代表类的Class对象的equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用instanceof关键字对对象所属关系的判定结果。

    站在Java虚拟机的角度来讲，只存在两种不同的类加载器：

* 启动类加载器：它使用C++实现（这里仅限于Hotspot，也就是JDK1.5之后默认的虚拟机，有很多其他的虚拟机是用Java语言实现的），是虚拟机自身的一部分。
* 所有其他的类加载器：这些类加载器都由Java语言实现，独立于虚拟机之外，并且全部继承自抽象类java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

    站在Java开发人员的角度来看，类加载器可以大致划分为以下三类：

* 启动类加载器：Bootstrap ClassLoader，跟上面相同。它负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.\*开头的类均被Bootstrap ClassLoader加载）。启动类加载器是无法被Java程序直接引用的。
* 扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载JDK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.\*开头的类），开发者可以直接使用扩展类加载器。
* 应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

     应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。因为JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件，因此如果编写了自己的ClassLoader，便可以做到如下几点：

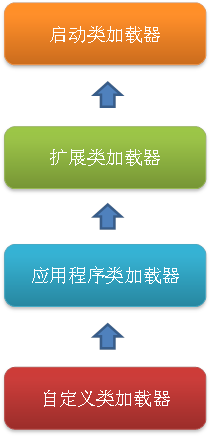
 1）在执行非置信代码之前，自动验证数字签名。

 2）动态地创建符合用户特定需要的定制化构建类。

 3）从特定的场所取得java class，例如数据库中和网络中。

事实上当使用Applet的时候，就用到了特定的ClassLoader，因为这时需要从网络上加载java class，并且要检查相关的安全信息，应用服务器也大都使用了自定义的ClassLoader技术。

    这几种类加载器的层次关系如下图所示：



    这种层次关系称为类加载器的双亲委派模型。我们把每一层上面的类加载器叫做当前层类加载器的父加载器，当然，它们之间的父子关系并不是通过继承关系来实现的，而是使用组合关系来复用父加载器中的代码。该模型在JDK1.2期间被引入并广泛应用于之后几乎所有的Java程序中，但它并不是一个强制性的约束模型，而是Java设计者们推荐给开发者的一种类的加载器实现方式。

    双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

    使用双亲委派模型来组织类加载器之间的关系，有一个很明显的好处，就是Java类随着它的类加载器（说白了，就是它所在的目录）一起具备了一种带有优先级的层次关系，这对于保证Java程序的稳定运作很重要。例如，类java.lang.Object类存放在JDK\jre\lib下的rt.jar之中，因此无论是哪个类加载器要加载此类，最终都会委派给启动类加载器进行加载，这边保证了Object类在程序中的各种类加载器中都是同一个类。

##### 验证

    验证的目的是为了确保Class文件中的字节流包含的信息符合当前虚拟机的要求，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

* 文件格式的验证：验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。经过该阶段的验证后，字节流才会进入内存的方法区中进行存储，后面的三个验证都是基于方法区的存储结构进行的。
* 元数据验证：对类的元数据信息进行语义校验（其实就是对类中的各数据类型进行语法校验），保证不存在不符合Java语法规范的元数据信息。
* 字节码验证：该阶段验证的主要工作是进行数据流和控制流分析，对类的方法体进行校验分析，以保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。
* 符号引用验证：这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候（解析阶段中发生该转化，后面会有讲解），主要是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。

##### 准备

    准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

    1、这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

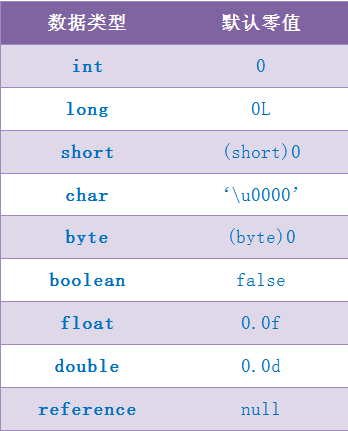
    2、这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

   假设一个类变量的定义为：

public static int value = 3；

    那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，而把value赋值为3的putstatic指令是在程序编译后，存放于类构造器<clinit>（）方法之中的，所以把value赋值为3的动作将在初始化阶段才会执行。

    下表列出了Java中所有基本数据类型以及reference类型的默认零值：



   这里还需要注意如下几点：

* 对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。
* 对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。
* 对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。
* 如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

    3、如果类字段的字段属性表中存在ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。

   假设上面的类变量value被定义为：

public static final int value = 3；

    编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。回忆[**上一篇博文**](http://blog.csdn.net/ns_code/article/details/17845821)中对象被动引用的第2个例子，便是这种情况。我们可以理解为static final常量在编译期就将其结果放入了调用它的类的常量池中。

##### 解析

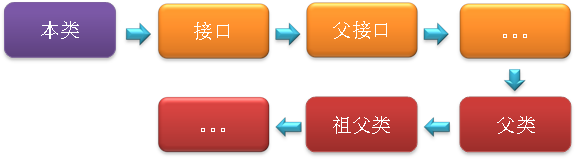
   解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。在[**Class类文件结构**](http://blog.csdn.net/ns_code/article/details/17675609)一文中已经比较过了符号引用和直接引用的区别和关联，这里不再赘述。前面说解析阶段可能开始于初始化之前，也可能在初始化之后开始，虚拟机会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析（初始化之前），还是等到一个符号引用将要被使用前才去解析它（初始化之后）。

    对同一个符号引用进行多次解析请求时很常见的事情，虚拟机实现可能会对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标示为已解析状态），从而避免解析动作重复进行。

    解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info四种常量类型。

**1、类或接口的解析：**判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。

**2、字段解析：**对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



   从下面一段代码的执行结果中很容易看出来字段解析的搜索顺序：

class Super{

public static int m = 11;

static{

System.out.println("执行了super类静态语句块");

}

}

class Father extends Super{

public static int m = 33;

static{

System.out.println("执行了父类静态语句块");

}

}

class Child extends Father{

static{

System.out.println("执行了子类静态语句块");

}

}

public class StaticTest{

public static void main(String[] args){

System.out.println(Child.m);

}

}

    执行结果如下：

    执行了super类静态语句块  
    执行了父类静态语句块  
    33  
    如果注释掉Father类中对m定义的那一行，则输出结果如下：

    执行了super类静态语句块  
    11

   另外，很明显这就是[**上篇博文**](http://blog.csdn.net/ns_code/article/details/17845821)中的第1个例子的情况，这里我们便可以分析如下：static变量发生在静态解析阶段，也即是初始化之前，此时已经将字段的符号引用转化为了内存引用，也便将它与对应的类关联在了一起，由于在子类中没有查找到与m相匹配的字段，那么m便不会与子类关联在一起，因此并不会触发子类的初始化。

    最后需要注意：理论上是按照上述顺序进行搜索解析，但在实际应用中，虚拟机的编译器实现可能要比上述规范要求的更严格一些。如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译。如果对上面的代码做些修改，将Super改为接口，并将Child类继承Father类且实现Super接口，那么在编译时会报出如下错误：

StaticTest.java:24: 对 m 的引用不明确，Father 中的 变量 m 和 Super 中的 变量 m  
都匹配  
                System.out.println(Child.m);  
                                        ^  
1 错误

**3、类方法解析：**对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。

**4、接口方法解析：**与类方法解析步骤类似，知识接口不会有父类，因此，只递归向上搜索父接口就行了。

##### 初始化

    初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码。在准备阶段，类变量已经被赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序指定的主观计划去初始化类变量和其他资源，或者可以从另一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。

   这里简单说明下<clinit>（）方法的执行规则:

    1、<clinit>（）方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句中可以赋值，但是不能访问。

    2、<clinit>（）方法与实例构造器<init>（）方法（类的构造函数）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>（）方法执行之前，父类的<clinit>（）方法已经执行完毕。因此，在虚拟机中第一个被执行的<clinit>（）方法的类肯定是java.lang.Object。

    3、<clinit>（）方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成<clinit>（）方法。

    4、接口中不能使用静态语句块，但仍然有类变量（final static）初始化的赋值操作，因此接口与类一样会生成<clinit>（）方法。但是接口鱼类不同的是：执行接口的<clinit>（）方法不需要先执行父接口的<clinit>（）方法，只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>（）方法。

5、虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>（）方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>（）方法完毕。如果在一个类的<clinit>（）方法中有耗时很长的操作，那就可能造成多个线程阻塞，在实际应用中这种阻塞往往是很隐蔽的。

总结

   整个类加载过程中，除了在加载阶段用户应用程序可以自定义类加载器参与之外，其余所有的动作完全由虚拟机主导和控制。到了初始化才开始执行类中定义的Java程序代码（亦及字节码），但这里的执行代码只是个开端，它仅限于<clinit>（）方法。类加载过程中主要是将Class文件（准确地讲，应该是类的二进制字节流）加载到虚拟机内存中，真正执行字节码的操作，在加载完成后才真正开始。

#### Java对象的访问方式

一般来说，一个Java的引用访问涉及到3个内存区域：JVM栈，堆，方法区。

Object obj = new Object(int a)为例：

Object obj表示一个引用，存储在JVM栈中，表示一个reference类型数据；

int a，基本数据类型，存储在栈中；

new Object()作为实例对象数据存储在堆中；

堆中还记录了Object类的类型信息（接口、方法、field、对象类型等）的地址，这些地址所执行的数据存储在方法区中；

#### JVM构成？内存分哪几个区，每个区的作用是什么?

java虚拟机主要分为:

（1）方法区(即Perm Gen持久代)：线程共享

1。 即Perm Gen(持久代)，方法区中包含的都是在整个程序中永远唯一的元素，如类名，访问修饰符，常量池，字段描述，方法描述，static变量和即时编译器编译后的代码等数据。很少发生GC(废弃常量和无用的类)

2。 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

（2）堆内存区域：线程共享，在虚拟机启动时创建

1。java堆又分为年轻代(Minor GC)和年老代(Major GC/Full GC)，

--1。Young年轻代：新建对象就在Eden区；Young区被划分为三部分，Eden(伊甸)区:Survivor(幸存)区:Survivor(幸存)区=8:1:1。某一时刻只用一个Survivor区，另外一个留做垃圾收集时复制对象用，在Young变满，minor GC将存活对象移到空闲Survivor区间中，根据JVM的策略(采用年龄计数器，每次Minor GC增加1岁，默认15岁移至年老代)，在经过几次垃圾收集后，仍然存活于Survivor的对象将被移动到Tenured区间。

--2。Tenured年老代：对象最活跃的区域是年老代(生命周期较长)，在年轻代中经历了N次垃圾回收仍然存活的对象，将被放到年老代中。一般如果系统中用了application级别的缓存，缓存中的对象往往会被转移到这一区间；大对象(很长的字符串/数组)直接进入老年代。System。gc()强制执行的是Full Gc。

（3）栈内存区域:线程私有的，它的生命周期与线程相同

1。 即虚拟机栈，它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法返回地址等信息。每一个方法从调用开始至结束都对应一个栈帧在虚拟机栈中的入栈和出栈过程。

2。局部变量表，操作数栈，动态连接，返回地址，

--局部变量表：存放方法参数和方法内部定义局部变量，大小是固定的；变量有基本类型和引用类型，二者作为局部变量，都放在栈中，基本类型直接在栈中保存值，引用类型只保存一个指向堆区的指针，真正的对象在堆里。所有数学运算都是在栈中进行。

--操作数栈：后入先出，操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式。

--动态连接：每个栈帧包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为支持方法调用过程的动态连接。

--返回地址：正常完成出口和异常完成出口。恢复上层方法的局部变量表和操作数栈，把返回值压入调用者栈帧的操作数栈中。

3。 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。动态链接就是将常量池中的符号引用在运行期转化为直接引用。

StackOverflowError：在线程执行的过程中栈空间不够用，那么JVM就会抛出此异常，死递归造成。

OutOfMemoryError:如果JVM内存大小自动扩展到计算机本身内存大小时会抛OutOfMemoryError。

（4）PC计数器：线程私有的，它的生命周期与线程相同

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

（5）本地方法栈：线程私有的，它的生命周期与线程相同

本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

总结：

其中栈、程序计数器、本地方法栈三个区域随线程而生，随线程而灭。这几个区域内存分配和回收都具有确定性，在这几个区域内不需要考虑太多的回收问题。

Java堆和方法区则不一样。一个接口中的多个实现类需要的内存可能不一样，一个方法中大多数分支需要的内存也可能不一样，我们只有在程序处于运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分的内存。

#### 堆与栈的比较：

（1）堆解决的是数据存储的问题，即数据怎么放、放在哪儿。栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；

（2）堆则是所有线程共享的。在Java中一个线程就会相应有一个线程栈与之对应。

（3）对象存放在堆中，对象引用和基本类型存放在栈中；对象的属性存放在堆中，对象的方法存在栈中。

程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

在JVM中，静态属性保存在Stack指令内存区，动态属性保存在Heap数据内存区

**堆的大小（1/64--1/4）**

-Xms ：指定了JVM初始启动的内存，表示java虚拟机堆区内存初始内存分配的大小，通常为操作系统可用内存的1/64大小即可

-Xmx：指定JVM堆的最大内存，在JVM启动以后，会分配-Xmx参数指定大小的内存给JVM，但是不一定全部使用，JVM会根据-Xms参数来调节真正用于JVM的内存，表示java虚拟机堆区内存可被分配的最大上限，通常为操作系统可用内存的1/4大小

-Xmx与-Xms之差就是三个Virtual空间的总大小（年轻代、年老代、持久代）

-------------

-XX:newSize

表示新生代初始内存的大小，应该小于 -Xms的值

-XX:MaxnewSize

表示新生代可被分配的内存的最大上限；当然这个值应该小于 -Xmx的值

-Xmn

表示同时配置XX:newSize = -XX:MaxnewSize = -Xmn

-------------

-XX:PermSize

表示非堆区初始内存分配大小，其缩写为permanent size（持久化内存）

-XX:MaxPermSize

表示对非堆区分配的内存的最大上限

---------------------------------------------------------

#### 简述java垃圾回收机制?

在java中，程序员是不需要显式的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫描那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

#### 分代垃圾回收

不同的对象生命周期不同。与业务信息有关的对象生命周期较长，如Http请求中的session对象、线程、socket连接。程序运行过程中的临时变量生命周期较短，如String对象。把不同生命周期的对象放在不同代上，不同代上采用最合适它的垃圾回收方式进行回收。

#### 简述java内存分配与回收策率以及Minor GC和Major GC

垃圾回收器利用一系列软指针跟踪对象的各个引用，并将其映射关系存至对象表中

JVM中共划分为三个代：年轻代、年老代和持久代。

持久代主要存放Java类的类信息，与垃圾收集要收集的Java对象关系不大。年轻代（minor Gc ）和年老代（Full Gc 或Major GC）的划分是对垃圾收集影响较大的。

对象优先在堆的Eden区(年轻代)分配。大对象和长期存活的对象直接进入老年代。当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC。Minor Gc通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生Gc的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC，但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

Java中GC标准：对象不可达；对象不再使用。Java提供finalize()终止化对象，释放资源

1。给对象赋空值且不再调用；2。给对象重新分配内存空间。

#### 如何判断一个对象是否存活?(或者GC对象的判定方法)。有两种方法:

1。 引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”，将会被垃圾回收。

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A，B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2。可达性算法(引用链法)

该算法的思想是：从一个被称为GC Roots的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。

在java中可以作为GC Roots的对象有以下几种:

**虚拟机栈中引用的对象**

**方法区类静态属性引用的对象**

**方法区常量池引用的对象**

**本地方法栈JNI引用的对象**

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比不一定会被回收。当一个对象不可达GC Root时，这个对象并不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记。

如果对象在可达性分析中没有与GC Root的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行finalize()方法，那么这个对象将会放在一个称为F-Queue的队列中，虚拟机会触发一个Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果finalize()执行缓慢或者发生了死锁，那么就会造成F-Queue队列一直等待，造成了内存回收系统的崩溃。GC对处于F-Queue中的对象进行第二次被标记，这时，该对象将被移除”即将回收”集合，等待回收。

#### java中垃圾收集的方法有哪些?

（1）复制算法:

复制算法将可用内存按容量划分为相等的两部分，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后清掉第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一半的内存。

（2）标记-清除:

垃圾收集算法中最基础，思想是标记哪些要被回收的对象，然后统一回收。方法很简单但会有两个问题：1。效率不高，标记和清除的效率都很低；2。会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。

（3）标记-整理

该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

（4）分代收集 （复制算法+标记整理/清除）

虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。复制算法进行改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用标记-整理 或者 标记-清除。

#### 垃圾收集器

**垃圾收集器是内存回收算法的具体实现，Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别。Sun  HotSpot虚拟机1.6版包含了如下收集器：Serial、ParNew、Parallel Scavenge、CMS、Serial Old、Parallel Old。这些收集器以不同的组合形式配合工作来完成不同分代区的垃圾收集工作。**

#### 垃圾回收分析

在用代码分析之前，我们对内存的分配策略明确以下三点：

对象优先在Eden分配。

大对象直接进入老年代。

长期存活的对象将进入老年代。

  对垃圾回收策略说明以下两点：

新生代GC（Minor GC）：发生在新生代的垃圾收集动作，因为Java对象大多都具有朝生夕灭的特性，因此Minor GC非常频繁，一般回收速度也比较快。

老年代GC（Major GC/Full GC）：发生在老年代的GC，出现了Major GC，经常会伴随至少一次Minor GC。由于老年代中的对象生命周期比较长，因此Major GC并不频繁，一般都是等待老年代满了后才进行Full GC，而且其速度一般会比Minor GC慢10倍以上。另外，如果分配了Direct Memory，在老年代中进行Full GC时，会顺便清理掉Direct Memory中的废弃对象。

    下面我们来看如下代码:

public class SlotGc{

public static void main(String[] args){

byte[] holder = new byte[32\*1024\*1024];

System.gc();

}

}

    代码很简单，就是向内存中填充了32MB的数据，然后通过虚拟机进行垃圾收集。在Javac编译后，我们执行如下指令：java -verbose:gc SlotGc来查看垃圾收集的结果，得到如下输出信息：

    [GC 208K->134K(5056K), 0.0017306 secs]

    [Full GC 134K->134K(5056K), 0.0121194 secs]

    [Full GC 32902K->32902K(37828K), 0.0094149 sec

    注意第三行，“->”之前的数据表示垃圾回收前堆中存活对象所占用的内存大小，“->”之后的数据表示垃圾回收堆中存活对象所占用的内存大小，括号中的数据表示堆内存的总容量，0.0094149 sec 表示垃圾回收所用的时间。

    从结果中可以看出，System.gc(（）运行后并没有回收掉这32MB的内存，这应该是意料之中的结果，因为变量holder还处在作用域内，虚拟机自然不会回收掉holder引用的对象所占用的内存。

    我们把代码修改如下：

public class SlotGc{

public static void main(String[] args){

{

byte[] holder = new byte[32\*1024\*1024];

}

System.gc();

}

}

    加入花括号后，holder的作用域被限制在了花括号之内，因此，在执行System.gc（）时，holder引用已经不能再被访问，逻辑上来讲，这次应该会回收掉holder引用的对象所占的内存。但查看垃圾回收情况时，输出信息如下：

    [GC 208K->134K(5056K), 0.0017100 secs]

    [Full GC 134K->134K(5056K), 0.0125887 secs]

    [Full GC 32902K->32902K(37828K), 0.0089226 secs]

    很明显，这32MB的数据并没有被回收。下面我们再做如下修改：

public class SlotGc{

public static void main(String[] args){

{

byte[] holder = new byte[32\*1024\*1024];

holder = null;

}

System.gc();

}

}

    这次得到的垃圾回收信息如下：

    [GC 208K->134K(5056K), 0.0017194 secs]

    [Full GC 134K->134K(5056K), 0.0124656 secs]

    [Full GC 32902K->134K(37828K), 0.0091637 secs]

    说明这次holder引用的对象所占的内存被回收了。我们慢慢来分析。

    首先明确一点：holder能否被回收的根本原因是局部变量表中的Slot是否还存有关于holder数组对象的引用。

在第一次修改中，虽然在holder作用域之外进行回收，但是在此之后，没有对局部变量表的读写操作，holder所占用的Slot还没有被其他变量所复用（回忆[J**ava内存区域与内存溢出**](http://blog.csdn.net/ns_code/article/details/17565503)一文中关于Slot的讲解），所以作为GC Roots一部分的局部变量表仍保持者对它的关联。这种关联没有被及时打断，因此GC收集器不会将holder引用的对象内存回收掉。 在第二次修改中，在GC收集器工作前，手动将holder设置为null值，就把holder所占用的局部变量表中的Slot清空了，因此，这次GC收集器工作时将holder之前引用的对象内存回收掉了。

    当然，我们也可以用其他方法来将holder引用的对象内存回收掉，只要复用holder所占用的slot即可，比如在holder作用域之外执行一次读写操作。

    为对象赋null值并不是控制变量回收的最好方法，以恰当的变量作用域来控制变量回收时间才是最优雅的解决办法。另外，赋null值的操作在经过虚拟机JIT编译器优化后会被消除掉，经过JIT编译后，System.gc（）执行时就可以正确地回收掉内存，而无需赋null值。

#### 性能调优

    Java虚拟机的内存管理与垃圾收集是虚拟机结构体系中最重要的组成部分，对程序（尤其服务器端）的性能和稳定性有着非常重要的影响。性能调优需要具体情况具体分析，而且实际分析时可能需要考虑的方面很多，这里仅就一些简单常用的情况作简要介绍。

**我们可以通过给Java虚拟机分配超大堆（前提是物理机的内存足够大）来提升服务器的响应速度，但分配超大堆的前提是有把握把应用程序的Full GC频率控制得足够低，因为一次Full GC的时间造成比较长时间的停顿。控制Full GC频率的关键是保证应用中绝大多数对象的生存周期不应太长，尤其不能产生批量的、生命周期长的大对象，这样才能保证老年代的稳定。**

**Direct Memory在堆内存外分配，而且二者均受限于物理机内存，且成负相关关系，因此分配超大堆时，如果用到了NIO机制分配使用了很多的Direct Memory，则有可能导致Direct Memory的OutOfMemoryError异常，这时可以通过-XX:MaxDirectMemorySize参数调整Direct Memory的大小。**

除了Java堆和永久代以及直接内存外，还要注意下面这些区域也会占用较多的内存，这些内存的总和会受到操作系统进程最大内存的限制：

1、线程堆栈：可通过-Xss调整大小，内存不足时抛出StackOverflowError（纵向无法分配，即无法分配新的栈帧）或OutOfMemoryError（横向无法分配，即无法建立新的线程）。

2、Socket缓冲区：每个Socket连接都有Receive和Send两个缓冲区，分别占用大约37KB和25KB的内存。如果无法分配，可能会抛出IOException：Too many open files异常。关于Socket缓冲区的详细介绍参见我的[**Java网络编程系列中深入剖析Socket**](http://blog.csdn.net/ns_code/article/details/15813809)的几篇文章。

3、JNI代码：如果代码中使用了JNI调用本地库，那本地库使用的内存也不在堆中。

4、虚拟机和GC：虚拟机和GC的代码执行也要消耗一定的内存。

#### 简述java内存模型

java内存模型(JMM)是线程间通信的控制机制。JMM定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存(main memory)中，每个线程都有一个私有的本地内存(local memory)，本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

-------------------------------------------------------------------

#### 简述java类加载机制?

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的java类型。

**java类加载过程?假烟准接触**

类加载的五个过程：加载、验证、准备、解析、初始化。

（1）加载Loading

加载是类加载的第一个过程，完成三件事情：

1。 通过一个类的全限定名获取该类的二进制流。

2。 将该二进制流中的静态存储结构转化为方法区的运行时数据结构。

3。 在内存中生成该类的Class对象，作为该类的数据访问入口。

（2）验证Verification

验证是为了确保Class文件的字节流中的信息不回危害到虚拟机，完成四种验证:

1。 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型。

2。 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。

3。 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。

4。 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

（3）准备Preparation

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。

（4）解析Resolution

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

（5）初始化Initialization

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

（6）使用Using

（7）卸载Unloading

#### 类加载器双亲委派模型机制？

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

#### 什么是类加载器，类加载器有哪些?

实现通过类的全限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器：契阔戏子

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。

2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader。getSystemClassLoader()来获取它。

4. 用户自定义类加载器，通过继承 java。lang。ClassLoader类的方式实现。

#### 内存溢出

    下面给出个内存区域内存溢出的简单[**测试**](http://lib.csdn.net/base/softwaretest)方法



    这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制Java堆和方法区这两部分内存的最大值，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，剩下的内存便给了虚拟机栈和本地方法栈，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过减少最大堆和每个线程的栈容量来换取更多的线程。

    另外，由于Java堆内也可能发生内存泄露（Memory Leak），这里简要说明一下内存泄露和内存溢出的区别：

    内存泄露是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们new了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露，

    内存溢出是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限。

#### 对象实例化分析

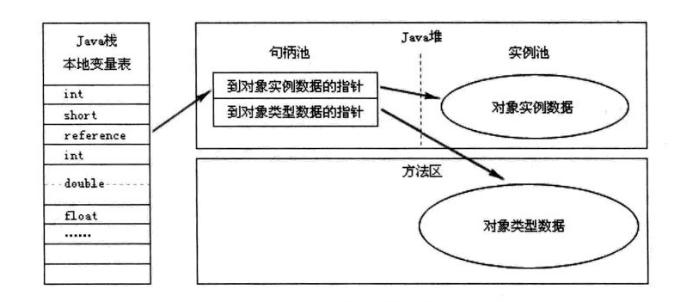
    对内存分配情况分析最常见的示例便是对象实例化:

**Object obj = new Object();**

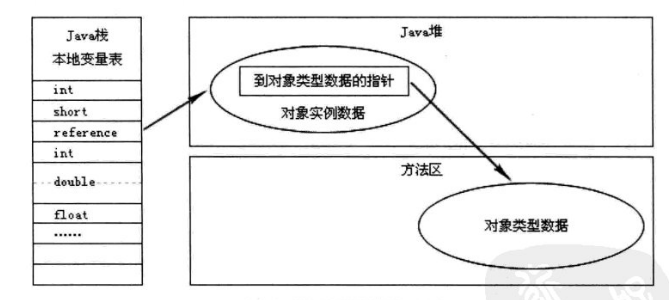
   这段代码的执行会涉及java栈、Java堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对JVM虚拟机不了解的Java使用这，应该也知道obj会作为引用类型（reference）的数据保存在Java栈的本地变量表中，而会在Java堆中保存该引用的实例化对象，但可能并不知道，Java堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。

    另外，由于reference类型在Java虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到Java堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄池和直接使用指针。

    通过句柄池访问的方式如下：



   通过直接指针访问的方式如下：



    这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是reference中存放的是稳定的句柄地址，在对象呗移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要修改。使用直接指针访问方式的最大好处是速度快，它节省了一次指针定位的时间开销。目前Java默认使用的HotSpot虚拟机采用的便是是第二种方式进行对象访问的。

#### Person p = new Person();

**创建一个对象都在内存中做了什么事情？**

1：先将硬盘上指定位置的Person。class文件加载进内存。

2：执行main方法时，在栈内存中开辟了main方法的空间(压栈-进栈)，然后在main方法的栈区分配了一个变量p。

3：在堆内存中开辟一个实体空间，分配了一个内存首地址值。new

4：在该实体空间中进行属性的空间分配，并进行了默认初始化。

5：对空间中的属性进行显示初始化。

6：进行实体的构造代码块初始化。

7：调用该实体对应的构造函数，进行构造函数初始化。（）

8：将首地址赋值给p ，p变量就引用了该实体。(指向了该对象)

代码编译：\*。java-->字节码。class-->解释器

Java的Class文件是有8个字节为基础的字搜索节流构成的，这些字节流之间都严格按照规定的顺序排列，并且字节之间不存在任何空隙，对于超过8个字节的数据，将按

照Big-Endian的顺序存储的，也就是说高位字节存储在低的地址上面，而低位字节存储到高地址上面，其实这也是class文件要跨平台的关键，因为

PowerPC架构的处理采用Big-Endian的存储顺序，而x86系列的处理器则采用Little-Endian的存储顺序，因此为了Class文

件在各中处理器架构下保持统一的存储顺序，虚拟机规范必须对起进行统一。

字节码文件是经过编译器预处理过的一种文件，是JAVA的执行文件存在形式，它本身是二进制文件，但是不可以被系统直接执行，而是需要虚拟机解释执行。

------------------------------------------------------------

#### Jar 创建和管理Java归档文件

Java java应用程序启动器。开发环境和生产环境都使用这个。jre命令废弃了。

Javac java程序语言编译器

Javadoc API文档生成器

Javah C头文件和stub生成器。用来写native的方法。

Javap 类文件反汇编程序。

Jdb java调试器

Jpeps java类依赖分析器

Jconsole 一个符合JMX规范的图形化工具，可以用来监控Java虚拟机。可以监控本地和远程的JVM。也可以监控和管理一个应用程序。

Jps 主要用来输出JVM中运行的进程状态信息。

Jstat 监控虚拟机运行状态，虚拟机类的加载与卸载管理内存的使用和垃圾收集等信息。

Jstack Java堆栈跟踪。打印一个给定的进程中一个线程、核心文件或者远程调试服务器的堆栈。

Jinfo 查看Java程序的配置信息。打印给定进程或者核心文件或者一个远程调试服务器的配置信息。

Jmap 查看等待回收对象的队列，查看堆的概要信息，生成jvm转储快照。Java的内存映射。打印共享的对象()内存映射或者给定进程、核心文件和远程调试服务器的堆栈内存细节。

#### 如何估算内存消耗

最佳内存配置需要捕获实时数据大小。实时数据的大小是由设置在其稳定状态运行应用程序所需的长期消耗对象的堆大小。

有了这个定义，我们在GC日志的打开的情况下就可以对应用程序进行负载测试（-XX：+PrintGCTimeStamps-Xloggc:/tmp/gc。log-XX:+PrintGCDetails）和可视化的日志（例如gcviewer）来确定应用程序达到稳定状态的时间。GC进行了minor GC和full GC。在大多数情况下，观察到变化的趋势需要10-20次full GC的运行。我们可以估算出实时数据大小。

JVM创建一个缓冲区的时候，实际上做了如下几件事：

1。JVM确保Heap区域内的空间足够，如果不够则使用触发GC在内的方法获得空间;

2。获得空间之后会找一组堆内的连续地址分配数组， 这里需要注意的是，在物理内存上，这些字节是不一定连续的;

对于不涉及到IO的操作，这样的处理没有任何问题，但是当进行IO操作的时候就会出现一点性能问题。

所有的IO操作都需要操作系统进入内核态才行，而JVM进程属于用户态进程， 当JVM需要把一个缓冲区写到某个Channel或Socket的时候，需要切换到内核态。

而内核态由于并不知道JVM里面这个缓冲区存储在物理内存的什么地址，并且这些物理地址并不一定是连续的(或者说不一定是IO操作需要的块结构)，所以在切换之前JVM需要把缓冲区复制到物理内存一块连续的内存上， 然后由内核去读取这块物理内存，整合成连续的、分块的内存。

JDK1。4之后引入了NIO， 提供了一种内存映射技术， 让我们可以直接从Java代码中创建DirectBuffer，这种Buffer在创建的时候直接就在物理内存中分配一块连续内存，当需要使用的时候不再需要复制，内核直接调用即可。 但缺点也是显而易见的，就是每次分配都比较昂贵一点，同时由于分配的内存不在Java Heap中，所以也不会受用户设置的堆大小的限制。

#### JVM 的特征包括：

JVM 是基于栈的虚拟机：最流行的计算机架构比如 Intel x86 架构和 ARM 架构是基于 寄存器 运行。然而 JVM 是基于栈运行。

符号引用：除了原始数据类型以外的所有类型（类和接口）都是通过符号引用，而不是通过直接的内存地址引用。

垃圾回收：类的实例是通过用户代码手动创建，但是通过垃圾回收自动销毁。

通过清晰的定义原始数据类型来保证平台独立性：传统的像 C/C++ 这样的语言在不同的平台上有不同的类型大小。JVM 通过清晰的定义原始数据类型来维持其兼容性，并且保证平台独立。

网络字节序：Java 类文件使用网络字节序。为了在Intel x86 架构的小端和 RISC 架构使用的大端之间维持平台独立，必须要使用一种混合的字节序。所以，JVM 使用网络字节序，网络字节序用于网络传输。网络字节序使用的是大端。

# 集合

Collection

（1）List：

ArrayList：随机存储(数组)，非线程安全，增加、删除、移动弱，随机访问元素强

Vector：随机存储(数组)，线程安全，增加、删除、移动弱，随机访问元素强

LinkedList：线性存储(链表)，增加、删除、移动强，随机访问元素弱

（2）Set值唯一

HashSet：未排序，性能优于TreeSet

TreeSet：排序，值唯一，实现SortedSet接口

LinkedHashSet:相比HashSet，多使用了链表维护元素的次序，迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

（3）Queue

priorityQueue：offer()添加元素，poll()删除元素

#### Map

（1）HashMap(数组)

底层是哈希表数据结构。线程不同步。允许null键null值，非线程安全，效率高

（2）HashTable

与HashMap比较，不允许null键null值，线程安全

（3）TreeMap(二叉树)

底层是二叉树数据结构，线程不同步，可用于给Map集合中的键进行排序。

#### AQS

AQS（抽象的队列式的同步器），AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它。

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

自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可：

isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。

<http://blog.csdn.net/defonds/article/details/44021605/>



java.util.concurrent - Java 并发工具包

Executor                 ：具体Runnable任务的执行者。

ExecutorService           线程池管理者，其实现类有多种，Runnable，Callable提交到池中让其调度。

ScheduledExecutorService ：安排在给定的延迟后运行或定期执行的命令。

Semaphore                 ：计数信号量

ReentrantLock             ：可重入的互斥锁定 Lock，功能类似synchronized，但要强大的多。

Future                    ：与Runnable，Callable进行交互的接口，线程执行结束后取返回的结果

BlockingQueue             ：阻塞队列。

CompletionService         : ExecutorService的扩展，可以获得线程执行结果的

CountDownLatch            ：在完一组正在其他线程中执行的操作之前，它允许一直等待

CyclicBarrier             ：栅栏，它允许一组线程互相等待，直到到达某个公共屏障点

#### Java.util.concurrent.locks:锁ReentrantLock/ReentrantReadWriteLock

1 BlockingQueue阻塞队列

主要方法：

作用:BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。

BlockingQueue 接口的实现：

(1)ArrayBlockingQueue

ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

ArrayBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

(2)DelayQueue

作用:DelayQueue 将会在每个元素的 getDelay() 方法返回的值的时间段之后才释放掉该元素。如果返回的是 0 或者负值，延迟将被认为过期，该元素将会在 DelayQueue 的下一次 take  被调用的时候被释放掉。

(3)LinkedBlockingQueue

作用:LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer。MAX\_VALUE 作为上限。

LinkedBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

(4)PriorityBlockingQueue

作用:PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java。util。PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。

所有插入到 PriorityBlockingQueue 的元素必须实现 java。lang。Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。

注意 PriorityBlockingQueue 对于具有相等优先级(compare() == 0)的元素并不强制任何特定行为。

(5)SynchronousQueue

作用:SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。

#### 2CountDownLatch闭锁

主要方法：countDown()await()

作用:CountDownLatch 以一个给定的数量初始化。countDown() 每被调用一次，这一数量就减一。通过调用 await() 方法之一，线程可以阻塞等待这一数量到达零。

#### 3CyclicBarrier栅栏

主要方法：await()

作用:CyclicBarrier 类是一种同步机制，它能够对处理一些算法的线程实现同步。即所有线程必须等待的一个栅栏，直到所有线程都到达这里，然后所有线程才可以继续做其他事情。

#### 4Exchanger交换机

主要方法：exchange()

作用:两个线程通过一个 Exchanger 交换对象。交换对象的动作由 Exchanger 的两个 exchange() 方法的其中一个完成。

#### 5Semaphore信号量

主要方法：acquire()release()

作用:计数信号量由一个指定数量的 "许可" 初始化。每调用一次 acquire()，一个许可会被调用线程取走。每调用一次 release()，一个许可会被返还给信号量。因此，在没有任何 release() 调用时，最多有 N 个线程能够通过 acquire() 方法，N 是该信号量初始化时的许可的指定数量。这些许可只是一个简单的计数器。

#### 5ExecutorService执行器服务接口

(1)ThreadPoolExecutor

ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定：corePoolSize，maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。

如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。

6ScheduledExecutorService定时执行者服务接口

(1)ScheduledThreadPoolExecutor

将任务延后执行，或者间隔固定时间多次执行。

7ForkJoinPool分叉和合并

配合 分叉-和-合并 任务分割的工作

RecursiveAction任何返回值的任务

RecursiveTask有返回值的任务

#### java.util.concurrent.locks.Lock

(1)ReentrantLock

主要方法：lock();lockInterruptibly();tryLock();tryLock(long timeout， TimeUnit timeUnit);unlock();

(2)ReentrantReadWriteLock

它能够允许多个线程在同一时间对某特定资源进行读取，但同一时间内只能有一个线程对其进行写入。

读写锁的理念在于多个线程能够对一个共享资源进行读取，而不会导致并发问题。并发问题的发生场景在于对一个共享资源的读和写操作的同时进行，或者多个写操作并发进行。

9java.util.concurrent.atomic原子性

(1)java.util.concurrent.atomic.AtomicBoolean原子性布尔

主要方法：get();set(false);getAndSet(false);compareAndSet(expectedValue， newValue);

(2)java.util.concurrent.atomic.AtomicInteger原子性整型

主要方法：get();set(12);getAndSet(12);compareAndSet(expectedValue， newValue);addAndGet(12);getAndAdd(12);getAndIncrement();incrementAndGet();

(3) java.util.concurrent.atomic .AtomicLong原子性长整型

主要方法：get();set(12);getAndSet(12);compareAndSet(expectedValue， newValue);addAndGet(12);getAndAdd(12);getAndIncrement();incrementAndGet();

(4) java.util.concurrent.atomic .AtomicReference原子性引用型

主要方法：get();set("New referenced");getAndSet("New referenced");compareAndSet(expectedValue， newValue);

java.util.concurrent.atomic 中的类可以分成4组：

标量类（Scalar）：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference

数组类：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray

更新器类：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

复合变量类：AtomicMarkableReference，AtomicStampedReference

保证原子性的原理：for循环先取得原来的值，做改变时与原来的值比较，值不同则不断尝试。

#### CAS:比较并交换

 public final int getAndIncrement() {

         \*        for (;;) {//死循环实现原子性

         \*              int current = get();  // 取得AtomicInteger里存储的数值

         \*            int next = current + 1;  // 加1

         \*            if (compareAndSet(current， next))   // 调用compareAndSet执行原子更新操作

         \*                return current;

         \*        }

         \*    }

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

public final boolean compareAndSet(int expect， int update) {

    return unsafe。compareAndSwapInt(this， valueOffset， expect， update);

    }

//native 方法，compareAndSwapInt 基于的是 CPU 的 CAS 指令来实现的

public final native boolean compareAndSwapInt(Object var1， long var2， int var4， intvar5);

---

#### ThreadPoolExecutor

public ThreadPoolExecutor(int corePoolSize，

                          int maximumPoolSize，

                          long keepAliveTime，

                          TimeUnit unit，

                          BlockingQueue<Runnable> workQueue，

                          ThreadFactory threadFactory，

                          RejectedExecutionHandler handler)

int corePoolSize//线程池维护线程的最少数量

int maximumPoolSize//线程池维护线程的最大数量

long keepAliveTime//线程池维护线程所允许的空闲时间

TimeUnit unit//线程池维护线程所允许的空闲时间的单位

BlockingQueue<Runnable> workQueue//线程池所使用的缓冲队列

ThreadFactory threadFactory，

RejectedExecutionHandler handler//线程池对拒绝任务的处理策略

1）当池子大小 小于 corePoolSize，就新建线程处理请求

2）当池子大小 等于 corePoolSize，请求放入workQueue中，池子里的空闲线程就去从workQueue中取任务并处理

3）当 workQueue 放不下新入的任务时，新建线程入池，并处理请求，如果池子大小撑到了maximumPoolSize就用RejectedExecutionHandler来做拒绝处理

4）当池子的线程数大于corePoolSize的时候，多余的线程会等待keepAliveTime长的时间，如果无请求可处理就自行销毁

ThreadPoolExecutor就是依靠BlockingQueue的阻塞机制来维持线程池，当池子里的线程无事可干的时候就通过workQueue.take()阻塞住

newCachedThreadPool

//newCachedThreadPool比较适合没有固定大小并且比较快速就能完成的小任务，没必要维持一个Pool，这比直接new Thread来处理的好处是能在60秒内重用已创建的线程。

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0，

Integer.MAX\_VALUE，

                                  60L，

TimeUnit.SECONDS，

                                  new SynchronousQueue<Runnable>());

//SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素的进出，线程会阻塞

}

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：工作线程的创建数量几乎没有限制(其实也有限制的，数目为Interger。 MAX\_VALUE)， 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

(2)newFixedThreadPool

public static ExecutorService newFixedThreadPool(int nThreads) {

    return new ThreadPoolExecutor(nThreads， nThreads，

                                  0L， TimeUnit.MILLISECONDS，

                                  new LinkedBlockingQueue<Runnable>());

//LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。可自定义上限。如果无自定义上限，将使用 Integer。MAX\_VALUE

}

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

(3)newSingleThreadExecutor

public static ExecutorService newSingleThreadExecutor() {

    return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1， 1，

                                    0L， TimeUnit.MILLISECONDS，

                                    new LinkedBlockingQueue<Runnable>()));

}

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO， LIFO， 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

(4)newScheduleThreadPool

public ScheduledThreadPoolExecutor(int corePoolSize) {

        super(corePoolSize， Integer。MAX\_VALUE， 0， TimeUnit。NANOSECONDS，

              new DelayedWorkQueue());

}

---

Java.util.concurrent。locks:锁ReentrantLock/ReentrantReadWriteLock

（1）ReentrantLock

class ReentrantLock implements Lock， java。io。Serializable

ReentrantLock获取锁定与三种方式：即interface Lock 接口的定义

lock()， 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁

tryLock()， 如果获取了锁立即返回true，如果别的线程正持有锁，立即返回false；

tryLock(long timeout，TimeUnit unit)，   如果获取了锁定立即返回true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false；

lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到获取锁定，或者当前线程被别的线程中断

static abstract class Sync extends AbstractQueuedSynchronizer

ReentrantLock把所有Lock接口的操作都委派到一个Sync类上

Sync又有两个子类：

final static class NonfairSync extends Sync

final static class FairSync extends Sync

显然是为了ReentrantLock支持公平锁和非公平锁而定义，默认情况下为非公平锁。

（2）ReentrantReadWriteLock

private Lock readLock = rwl。readLock();//得到可被多个读操作共用的读锁，但它会排斥所有写操作

private Lock writeLock = rwl。writeLock();//得到写锁，它会排斥所有其他的读操作和写操作

-------------------------------------

#### volatile特性：

**可见性：对volatile变量读，总能看到对这个volatile的写入（写-读基于happens-before）**

**原子性：对任意单个volatile变量的读/写具有原子性，但类似volatile++这种复合操作不具有原子性**

**jvm底层采用内存屏障来实现volatile语义，被volatile修饰的变量禁止编译器和处理器重排序操作**

**volatile[ˈvɑ:lətl]与synchronized['sɪŋkrənaɪzd]**

**1）volatile本质告诉jvm当前变量在寄存器中的值是不确定的，都强迫从共享内存中重读该成员变量的值(当成员变量发生变化时，强迫线程将变化值回写到共享内存)；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。**

**2）volatile仅能使用在变量级别，仅能实现变量的修改可见性；synchronized则可以使用在变量，方法。保证变量的修改可见性和原子性**

**3）volatile不会造成线程的阻塞，而synchronized可能会造成线程的阻塞。**

#### ReentrantLock 与synchronized 区别

1）ReentrantLock 拥有Synchronized相同的并发性和内存语义，还多了锁投票，定时锁等候和中断锁等候。synchronized会造成线程的阻塞。ReentrantLock尝试获取不到锁中断去干别的事情。

2）synchronized可用在方法、对象、常量上，是在JVM层面上实现的，可通过监控工具监控synchronized的锁定，在代码执行时出现异常JVM会自动释放锁定。但Lock不行，lock是通过应用代码实现的，必须在finally中释放锁。

3）在资源竞争不激烈的情况下，Synchronized的性能要优于ReetrantLock，但在资源竞争激烈，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维持常态；

synchronized 对象锁和类锁的区别

pulbic class Something(){

    public synchronized void isSyncA(){}

    public static synchronized void cSyncB(){}

}

一个锁的是类对象(cSyncB)，一个锁的是实例对象(isSyncA)。

若类对象被lock，则类对象的所有同步方法全被lock；

若类的某一实例对象被lock，则该实例对象的所有同步方法全被lock，但是该类的另一实例对象不受影响。

synchronized关键字在经过编译之后会在同步块的前后分别形成monitor enter 和minitor exit 这两个字节码指令。

#### 死锁原因：

产生死锁的原因主要是：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则

就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之

一不满足，就不会发生死锁。

死锁的解除与预防：

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和

解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确

定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态

的情况下占用资源。因此，对资源的分配要给予合理的规划。

死锁是两个甚至多个线程被永久阻塞时的一种运行局面，这种局面的生成伴随着至少两个线程和两个或者多个资源。

使用VisualVM或者jstack命令产生线程转存，线程转存的输出可以清楚地说明死锁环境和线程，以及包含死锁环境的资源(每一个资源都有一个独特的ID)。

**避免死锁:**

避免嵌套封锁：死锁最主要原因，如果你已经有一个资源了就要避免封锁另一个资源。如果你运行时只有一个对象封锁，那是几乎不可能出现一个死锁局面的。

减小锁定范围：只对有请求的进行封锁，你应当只想你要运行的资源获取封锁，并非完全的对象。

避免无限期等待：如果两个线程正在等待对象结束，无限期的使用线程加入，如果你的线程必须要等待另一个线程的结束，若是等待进程的结束加入最好准备最长时间。

多线程上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

#### Runnable和Callable的区别

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；

Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

**FutureTask**

FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

#### 一个线程如果出现了运行时异常怎么办?

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放。

#### 如何在两个线程间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

-------------------------------------------------------

#### HashMap+ConcurrentHashMap+HashTable+HashSet

Java.util.HashMap

<http://www.cnblogs.com/ITtangtang/p/3948406.html>

除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。

实现线程安全：Map map = Collections.synchronizedMap(new HashMap());

HashMap

HashMap其实就是一个Entry数组，Entry对象中包含了键和值，其中next也是一个Entry对象，它就是用来处理hash冲突的，形成一个链表。

HashMap如何解决指针碰撞问题：HashMap的底层主要是基于数组和链表来实现的，查询速度快因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。HashMap底层是通过链表来解决hash冲突的。

    关键属性

transient Entry[] table;//存储元素的实体数组

transient int size;//存放元素的个数

int threshold; //临界值   当实际大小超过临界值时，会进行扩容threshold = 加载因子\*容量

final float loadFactor; //负载因子，表示Hsah表中元素的填满的程度

transient int modCount;//被修改的次数

    默认值

static final int DEFAULT\_INITIAL\_CAPACITY = 16;//数组默认容量

static final int MAXIMUM\_CAPACITY = 1 << 30;//数组最大容量

static final float DEFAULT\_LOAD\_FACTOR = 0。75f;//默认负载因子

    覆盖的方法：来自 Object

equals(Object o)比较指定对象与此 Map 的等价性<HashTable synchronized>

hashCode()返回此 Map 的哈希码<HashTable synchronized>

clone() 复制对象<HashTable synchronized>

toString() 获取字符串<HashTable synchronized>

    Map 更新方法： 可以更改 Map 内容

remove(Object key)从 Map 中删除键和关联的值<HashTable synchronized>

put(Object key， Object value)将指定值与指定键相关联<HashTable synchronized>

clear()从 Map 中删除所有映射<HashTable synchronized>

putAll(Map t)将指定 Map 中的所有映射复制到此 map<HashTable synchronized>

    返回视图的 Map 方法：可以遍历 Map 的元素，删除 Map 中的元素。

entrySet()返回 Map 中所包含映射的 Set 视图。Set 中的每个元素都是一个 Map。Entry 对象，可以使getKey() / getValue() 方法（还有一个 setValue() 方法）访问后者的键元素和值元素

keySet()返回 Map 中所包含键的 Set 视图。删除 Set 中的元素还将删除 Map 中相应的映射（键和值）

values()返回 map 中所包含值的 Collection 视图。删除 Collection 中的元素还将删除 Map 中相应的映射（键和值）

    访问元素： 不更改 Map 内容

get(Object key)返回与指定键关联的值<HashTable synchronized>

containsKey(Object key)Map 包含指定键的映射则 true<HashTable synchronized>

containsValue(Object value) Map 将一个或多个键映射到指定值则 true<HashTable synchronized>

isEmpty()如果 Map 不包含键-值映射，则返回 true<HashTable synchronized>

size()返回 Map 中的键-值映射的数目<HashTable synchronized>

---

构造方法

HashMap()：构建初始容量 16，负载因子 0。75 。

HashMap(int initialCapacity)：初始容量 initialCapacity，负载因子 0。75

HashMap(int initialCapacity， float loadFactor)：以指定初始容量、指定的负载因子创建

数组初始容量initialCapacity的巧妙设计：确保容量为2的n次幂

int capacity = 1;   //初始容量

         while (capacity < initialCapacity) {

//确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂

             capacity <<= 1;}

---

//计算存储地址:h&(length-1)就相当于对length取模，保证散列的均匀，同时也提升了效率。

与的计算规则是，两个都为真(或为1)，结果为真，如果两位有一位为假(或为0）结果为假

HashMap哈希函数取数组地址：

static int indexFor(int h， int length) { //根据hash值和数组长度算出索引值

         return h & (length-1);  //hash&(length-1)可以确保算出来的索引是在数组大小范围内，不会超出

     }

Hashtable采用hash值对length取模（即除法散列法）实现，但取模会用到除法运算，效率很低

---

存储数据

public V put(K key， V value) {

     // 若“key为null”，则将该键值对添加到table[0]中。

         if (key == null)

            return putForNullKey(value);

     // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。

         int hash = hash(key。hashCode());

     //搜索指定hash值在对应table中的索引

         int i = indexFor(hash， table。length);

     // 循环遍历Entry数组，若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！

         for (Entry<K，V> e = table[i]; e != null; e = e。next) {

             Object k;

              if (e。hash == hash && ((k = e。key) == key || key。equals(k))) { //如果key相同则覆盖并返回旧值

                  V oldValue = e。value;

                 e。value = value;

                 e。recordAccess(this);

                 return oldValue;

              }

         }

     //修改次数+1

         modCount++;

     //将key-value添加到table[i]处

     addEntry(hash， key， value， i);

     return null;

}

---

//addEntry() 方法发生冲突时在链表新增

void addEntry(int hash， K key， V value， int bucketIndex) {

         Entry<K，V> e = table[bucketIndex];

//如果要加入的位置有值，将该位置原先的值设置为新entry的next，也就是新entry链表的下一个节点

         table[bucketIndex] = new Entry<>(hash， key， value， e);

         if (size++ >= threshold) //如果大于临界值就扩容

             resize(2 \* table.length); //以2的倍数扩容

     }

---

数组实体对象Entry

Entry是单向链表，构造函数参数包括"哈希值(h)"， "键(k)"， "值(v)"， "下一节点(n)"

        Entry(int h， K k， V v， Entry<K，V> n) {

            value = v;

            next = n;

            key = k;

            hash = h;

        }

//若两个Entry的“key”和“value”都相等，则返回true，否则，返回false

---

resize()方法，重新调整HashMap的大小，newCapacity是调整后的单位

void resize(int newCapacity) {

          Entry[] oldTable = table;//传入新的容量

          int oldCapacity = oldTable。length;//引用扩容前的Entry数组

          if (oldCapacity == MAXIMUM\_CAPACITY) {//扩容前的数组大小如果已经达到最大(2^30)了

              threshold = Integer.MAX\_VALUE;//修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

              return;

         }

         Entry[] newTable = new Entry[newCapacity];//初始化一个新的Entry数组

         transfer(newTable);//用来将原先table的元素全部移到newTable里面

         table = newTable;  //再将newTable赋值给table

         threshold = (int)(newCapacity \* loadFactor);//重新计算临界值

}

---

读取数据

public V get(Object key) {

    if (key == null)

        return getForNullKey();

    int hash = hash(key。hashCode());

    for (Entry<K，V> e = table[indexFor(hash， table.length)];

        e != null;

        e = e.next) {

        Object k;

        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

            return e.value;

    }

    return null;

}

---

//关键属性 table  size transient原因：数据中空余空间实例化没有意义；不同虚拟机的hashCode值可能不同引发错误

HashMap自己重写了readObject和writeObject实现数据序列化

1.transient 是表明该数据不参与序列化.因为 HashMap 中的存储数据的数组数据成员中，数组还有很多的空间没有被使用，没有被使用到的空间被序列化没有意义。所以需要手动使用 writeObject() 方法，只序列化实际存储元素的数组。

2。 由于不同的虚拟机对于相同 hashCode 产生的 Code 值可能是不一样的，如果你使用默认的序列化，那么反序列化后，元素的位置和之前的是保持一致的，可是由于 hashCode 的值不一样了，那么定位函数 indexOf（）返回的元素下标就会不同，这样不是我们所想要的结果。

---

#### HashMap什么时候进行扩容呢？

当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，扩容是需要进行数组复制的，复制数组是非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

负载因子衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

---

#### HashMap死锁原因：

多线程环境下get()方法造成的死锁。肯定是put()方法引起的，resize()方法-->transfer()方法

transfer()转移的时候是逆序的，假如转移前链表顺序是1->2->3，那么转移后就会变成3->2->1

---------------------------------------------------------------

#### ConcurrentHashMap

HashMap是根据Hash值在计算索引存储在数组table中的，同步Map在同步的时候锁住了整个table，而ConcurrentHashMap加锁的时候根据散列值锁住了散列值锁对应的那段，因此提高了并发性能。

ConcurrentHashMap也增加了对常用复合操作的支持，比如"若没有则添加"：putIfAbsent()，替换：replace()。这2个操作都是原子操作。

实体类三个：

ConcurrentHashMap（整个Hash表）：

---

Segment（段）：

//

final Segment<K，V>[] segments;

//定位段位置

final Segment<K，V> segmentFor(int hash) {

    return segments[(hash >>> segmentShift) & segmentMask];

 }

//Segment定义

static final class Segment<K，V> extends ReentrantLock implements Serializable {

    transient volatile int count;//Segment中元素的数量

    transient int modCount;//对table的大小造成影响的操作的数量（比如put或者remove操作）

    transient int threshold;//阈值，元素的数量超过这个值依旧就会对Segment进行扩容

    transient volatile HashEntry<K，V>[] table;//链表数组

    final float loadFactor;//负载因子，用于确定threshold

}

---

HashEntry（节点）：

static final class HashEntry<K，V> {

     final K key;

     final int hash;

     volatile V value;

     final HashEntry<K，V> next;

 }

//除了value不是final的，其它值都是final的，这意味着不能从hash链的中间或尾部添加或删除节点(防止链表结构被破坏)，因为这需要修改next 引用值，所有的节点的修改只能从头部开始。

将value设置成volatile，保证线程间的可见性，避免了get加锁。

对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作，所以当竞争发生的时候，volatile的语义可以保证写操作在读操作之前，也就保证了写操作对后续的读操作都是可见的。

#### ConcurrentHashMap的并发度是什么?

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势。

整个锁定

size()、isEmpty()和containsValue()方法需要跨段，需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。

size()操作避免全局加锁的诀窍：

size操作就是遍历了两次Segment，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，就将原先遍历的结果返回，如果不相同，则把这个过程再重复做一次，如果再不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了

ConcurrentMap.putIfAbsent()

//如果（调用该方法时）key-value 已经存在，返回原 value 。如果不存在，返回一个 null

Locale l = cache.putIfAbsent(key， locale);

if (l != null) {

    locale = l;

}

---------------------------------------------------------------

#### HashTable

不允许使用null，synchronized是针对整张Hash表的，即每次锁住整张表让线程独占。对HashMap相对应各个方法加锁实现同步。

#### HashMap/HashTable/ConcurrenHashMap

HashMap与HashTable的区别是什么?

1.HashTable基于Dictionary类，而HashMap是基于AbstractMap。Dictionary是任何可将键映射到相应值的类的抽象父类，而AbstractMap是基于Map接口的实现，它以最大限度地减少实现此接口所需的工作。

2. HashMap的key和value都允许为null，而Hashtable的key和value都不允许为null。HashMap遇到key为null的时候，调用putForNullKey方法进行处理，而对value没有处理；Hashtable遇到null，直接返回NullPointerException。

3. HashMap是非同步的(HashMap未实现synchronized加锁，所以在addEntry、removeEntryForKey、resize三方法时会脏读，脏写)，也可以通过Collections.synchronizedMap(hashMap)，使其实现同步。

HashMap是无序的

#### HashMap的工作原理是什么?

HashMap内部是通过一个数组实现的，只是这个数组比较特殊，数组里存储的元素是一个Entry实体(jdk 8为Node)，这个Entry实体主要包含key、value以及一个指向自身的next指针。HashMap是基于hashing实现的，当我们进行put操作时，根据传递的key值得到它的hashcode，然后再用这个hashcode与数组的长度进行模运算，得到一个int值，就是Entry要存储在数组的位置（下标）；当通过get方法获取指定key的值时，会根据这个key算出它的hash值（数组下标），根据这个hash值获取数组下标对应的Entry，然后判断Entry里的key，hash值或者通过equals()比较是否与要查找的相同，如果相同，返回value，否则的话，遍历该链表（有可能就只有一个Entry，此时直接返回null），直到找到为止，否则返回null。

HashMap之所以在每个数组元素存储的是一个链表，是为了解决hash冲突问题，当两个对象的hash值相等时，那么一个位置肯定是放不下两个值的，于是hashmap采用链表来解决这种冲突，hash值相等的两个元素会形成一个链表。

HashMap有序的Map有LinkedHashMap、TreeMap

#### ConcurrenHashMap的工作原理?ConcurrenHashMap与Hashtable区别？

都是线程安全的，实现线程安全的方式不同。

ConcurrenHashMap可以说是HashMap的升级版，底层是“数组+链表->红黑树”的实现。

(表锁定-阻塞：分离锁-局部阻塞)Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁，局部锁定。

具体实现:

ConcurrentHashMap内部有一个Segment数组，该Segment对象可以充当锁。Segment对象内部有一个HashEntry数组，于是每个Segment可以守护若干个桶(HashEntry)，每个桶又有可能是一个HashEntry连接起来的链表，存储发生碰撞的元素。

每个ConcurrentHashMap在默认并发级下会创建包含16个Segment对象的数组，每个数组有若干个桶，当我们进行put方法时，通过hash方法对key进行计算，得到hash值，找到对应的segment，然后对该segment进行加锁，然后调用segment的put方法进行存储操作，此时其他线程就不能访问当前的segment，但可以访问其他的segment对象，不会发生阻塞等待。

#### ConcurrentHashMap

（1）ConcurrentHashMap的锁分段技术

ConcurrentHashMap采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size()， containsValue()）需要使用特殊的实现，另外一些方法（如clear()）甚至放弃了对一致性的要求（ConcurrentHashMap是弱一致性的)

（2）ConcurrentHashMap的读是否要加锁，为什么

分段锁机制：使用Segment来实现减小锁粒度，把HashMap分割成若干个Segment(默认的并发度为16)，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

（3）ConcurrentHashMap的迭代器是强一致性的迭代器还是弱一致性的迭代器

get不需加锁(可能取不到同时新增的数据)，ConcurrentHashMap是弱一致性的。

#### HashSet

一、  HashSet概述：

   HashSet实现Set接口，由哈希表（实际上是一个HashMap实例）支持。它不保证set 的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用null元素。

二、  HashSet的实现：

   对于HashSet而言，它是基于HashMap实现的，HashSet底层使用HashMap来保存所有元素，因此HashSet 的实现比较简单，相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成，

    HashSet的底层实现是什么?

通过看源码知道HashSet的实现是依赖于HashMap的，HashSet的值都是存储在HashMap中的。在HashSet的构造法中会初始化一个HashMap对象，**HashSet不允许值重复，因此，HashSet的值是作为HashMap的key存储在HashMap中的，当存储的值已经存在时返回false。**

---

java.util 程序包中实现

HashMap

Hashtable

Properties

LinkedHashMap

IdentityHashMap

TreeMap

WeakHashMap

ConcurrentHashMap

---

Hash构造函数的方法：

1.直接定址法：

2.数字分析法：

3.折叠法：

4.平方取中法

5.减去法

6.基数转换法

7.除留余数法：

8.随机数法：

9．随机乘数法

10．字符串数值哈希法

11.旋转法

Hash构造函数，通常应考虑以下五个因素 ：

l 计算哈希函数所需时间 （简单）

l 关键字的长度

l 哈希表大小

l 关键字分布情况

l 记录查找频率

哈希查找有两个步骤:

1. 使用哈希函数将被查找的键转换为数组的索引.

2. 处理哈希碰撞冲突.

Hash处理冲突方法

1. 开放定址法：

线性探测再散列（顺序查看表中下一单元）、二次探测再散列（表的左右进行跳跃式探测）、伪随机探测再散列

2。 再哈希法

3. 链地址法

4. 建立公共溢出区

将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

-------------------------------------------

#### ArrayList+Vector+CopyOnWriteArrayList

<http://www.cnblogs.com/ITtangtang/p/3948555.html>

ArrayList是基于数组实现的，是一个动态数组，其容量能自动增长

ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能被克隆。

私有属性:ArrayList只定义类两个私有属性

private transient Object[] elementData;  //存储ArrayList内的元素

private int size;//包含的元素的数量，默认初始容量为10

构造方法:三个

// ArrayList带容量大小的构造函数。

public ArrayList(int initialCapacity) {

        super();

        if (initialCapacity < 0)

            throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);

        // 新建一个数组

        this.elementData = new Object[initialCapacity];

}

元素存储：

set(int index， E element)

add(E e)

add(int index， E element)

addAll(Collection<? extends E> c)

addAll(int index， Collection<? extends E> c)

// 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。

public E set(int index， E element) {

   RangeCheck(index);

   E oldValue = (E) elementData[index];

   elementData[index] = element;

   return oldValue;

}

// 将指定的元素添加到此列表的尾部。

public boolean add(E e) {

   ensureCapacity(size + 1);

   elementData[size++] = e;

   return true;

}

// 将指定的元素插入此列表中的指定位置。

// 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。

public void add(int index， E element) {

   if (index > size || index < 0)

       throw new IndexOutOfBoundsException("Index: "+index+"， Size: "+size);

   // 如果数组长度不足，将进行扩容。

   ensureCapacity(size+1);  // Increments modCount!!

   // 将 elementData中从Index位置开始、长度为size-index的元素，

   // 拷贝到从下标为index+1位置开始的新的elementData数组中。

   // 即将当前位于该位置的元素以及所有后续元素右移一个位置。

   System.arraycopy(elementData， index， elementData， index + 1， size - index);

   elementData[index] = element;

   size++;

}

// 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。

public boolean addAll(Collection<? extends E> c) {

   Object[] a = c.toArray();

   int numNew = a.length;

   ensureCapacity(size + numNew);  // Increments modCount

   System.arraycopy(a， 0， elementData， size， numNew);

   size += numNew;

   return numNew != 0;

}

// 从指定的位置开始，将指定collection中的所有元素插入到此列表中。

public boolean addAll(int index， Collection<? extends E> c) {

   if (index > size || index < 0)

       throw new IndexOutOfBoundsException(

           "Index: " + index + "， Size: " + size);

   Object[] a = c.toArray();

   int numNew = a.length;

   ensureCapacity(size + numNew);  // Increments modCount

   int numMoved = size - index;

   if (numMoved > 0)

   System.arraycopy(elementData， index， elementData， index + numNew， numMoved);

   System.arraycopy(a， 0， elementData， index， numNew);

   size += numNew;

   return numNew != 0;

   }

元素读取：

// 返回此列表中指定位置上的元素。

 public E get(int index) {

    RangeCheck(index);

    return (E) elementData[index];

  }

元素删除：

// 移除此列表中指定位置上的元素。

 public E remove(int index) {

    RangeCheck(index);

    modCount++;

    E oldValue = (E) elementData[index];

    int numMoved = size - index - 1;

    if (numMoved > 0)

        System.arraycopy(elementData， index+1， elementData， index， numMoved);

    elementData[--size] = null; // Let gc do its work

    return oldValue;

 }

remove(Object o)

removeRange(int fromIndex，int toIndex)

调整数组容量ensureCapacity： int newCapacity = (oldCapacity \* 3)/2 + 1;  //增加50%+1

public void ensureCapacity(int minCapacity) {

    modCount++;

    int oldCapacity = elementData.length;

    if (minCapacity > oldCapacity) {

//局部变量oldData变量引用旧的内存块的话，在copy的过程中就会比较安全，因为这样证明这块老的内存依然有引用，分配内存的时候就不会被侵占掉

        Object oldData[] = elementData;

        int newCapacity = (oldCapacity \* 3)/2 + 1;  //增加50%+1

            if (newCapacity < minCapacity)

                newCapacity = minCapacity;

      // minCapacity is usually close to size， so this is a win:

      elementData = Arrays.copyOf(elementData， newCapacity);

    }

 }

1、注意其三个不同的构造方法。无参构造方法构造的ArrayList的容量默认为10，带有Collection参数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData。

2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用Arrays.copyof()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList。

3、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。

System.arraycopy()-->C语言的memmove()

4、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

5、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。

6、List<String> list = new ArrayList<>(2);//底层新建了个长度为2的空的数组Object[]

list.size();//size是表示这个集合包含的元素的个数，而不是底层数组的大小

#### CopyOnWrite容器

即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

CopyOnWriteArrayList是ArrayList 的一个线程安全的变体，其中所有可变操作（add、set等等）都是通过对底层数组进行一次新的复制来实现的。

**LinkedList和ArrayList的区别是什么?**

ArrayList是基于数组实现，LinkedList是基于链表实现

ArrayList在查找时速度快，LinkedList在插入与删除时更具优势

**Array和ArrayList有何区别？什么时候更适合用Array？**

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是固定的

---

ArrayList和Vector区别如下：

ArrayList在内存不够时默认是扩展50% + 1个，Vector是默认扩展1倍。

Vector提供indexOf(obj， start)接口，ArrayList没有。

Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为线程安全需要更大的系统开销。

---

**CopyOnWriteArrayList/CopyOnWriteArraySet**

//底层的数组采用Volatile来声明的，保证可见。

private volatile transient Object[] array;

基本思想是一旦对容器有修改，那么就“复制”一份新的集合，在新的集合上修改，然后将新集合复制给旧的引用。当然了这部分少不了要加锁。显然对于CopyOnWriteArrayList/CopyOnWriteArraySet来说最大的好处就是“读”操作不需要锁了。

读：get方法/contains/indexOf：不加锁

public E get(int index) {

        return get(getArray()， index);

    }

    final Object[] getArray() {

        return array;

    }

    private E get(Object[] a， int index) {

        return (E) a[index];

    }

public int indexOf(Object o) {

        Object[] elements = getArray();//得到此时数组的一份镜像

        return indexOf(o， elements， 0， elements.length);

    }

    private static int indexOf(Object o， Object[] elements，

                               int index， int fence) {

        if (o == null) {

            for (int i = index; i < fence; i++)

                if (elements[i] == null)

                    return i;

        } else {

            for (int i = index; i < fence; i++)

                if (o.equals(elements[i]))

                    return i;

        }

        return -1;

    }

    public boolean contains(Object o) {

        Object[] elements = getArray();

        return indexOf(o， elements， 0， elements.length) >= 0;

    }

可变操作(add/set/remove)

1、加锁

2、将原来的数组copy一份到新数组中，然后修改

3、将旧的引用array指向新数组。

4、释放锁

   public boolean add(E e) {

        final ReentrantLock lock = this.lock;

        lock.lock();

        try {

            Object[] elements = getArray();

            int len = elements.length;

            Object[] newElements = Arrays.copyOf(elements， len + 1);

            newElements[len] = e;

            setArray(newElements);//将新数组给原来的引用

            return true;

        } finally {

            lock.unlock();

        }

    }

#### foreach循环的原理

在foreach循环中，迭代集合collectionObject的过程如下：

（1）调用collectionObject.GetEnumerator()，返回一个IEnumerator引用。这个方法可以通过IEnumerable接口的实现代码来获得。但这是可选的。

（2）调用返回的IEnumerator接口的MoveNext()方法。

（3）如果MoveNext()方法返回true，就使用IEnumerator接口的Current属性获取对象的一个引用，用于foreach循环。

（4）重复前面两步，直到MoveNext()方法返回false为止，此时循环停止。

任何一个集合，无论是JDK提供的还是自己写的，只要想使用foreach循环遍历，就必须正确地实现Iterable接口

public interface Iterator

{

    public abstract boolean hasNext();

    public abstract Object next();

    public abstract void remove();

}

ArrayList、LinkedList、Hashtable、HashMap、ConcurrentHashMap、HashSet的实现原理

1、ArrayList之所以能使用foreach循环遍历，是因为ArrayList所有的List都是Collection的子接口，而Collection是Iterable的子接口，ArrayList的父类AbstractList正确地实现了Iterable接口的iterator方法。

**遍历一个List有哪些不同的方式？**

    List<String> strList = new ArrayList<>();

    //for-each

    for(String str:strList) {

        System.out.print(str);

    }

    //use iterator 尽量使用这种 更安全(fail-fast)

    Iterator<String> it = strList.iterator();

    while(it.hasNext) {

        System.out.printf(it.next());

    }

#### fail-fast与fail-safe有什么区别？

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。

Java.util包中的所有集合类都被设计为fail->fast的，

java.util.concurrent中的集合类都为fail-safe的。当检测到正在遍历的集合的结构被改变时，Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛异常。

#### LinkedHashMap的实现原理?

LinkedHashMap也是基于HashMap实现的，不同的是它定义了一个Entry header，这个header不是放在Table里，它是额外独立出来的。LinkedHashMap通过继承hashMap中的Entry，并添加两个属性Entry before，after，和header结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

#### happen—before规则介绍

[**Java**](http://lib.csdn.net/base/java)语言中有一个“先行发生”（happen—before）的规则，它是Java内存模型中定义的两项操作之间的偏序关系，如果操作A先行发生于操作B，其意思就是说，在发生操作B之前，操作A产生的影响都能被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等，它与时间上的先后发生基本没有太大关系。这个原则特别重要，它是判断数据是否存在竞争、线程是否安全的主要依据。

    举例来说，假设存在如下三个线程，分别执行对应的操作:

---------------------------------------------------------------------------

线程A中执行如下操作：i=1

线程B中执行如下操作：j=i

线程C中执行如下操作：i=2

---------------------------------------------------------------------------

    假设线程A中的操作”i=1“ happen—before线程B中的操作“j=i”，那么就可以保证在线程B的操作执行后，变量j的值一定为1，即线程B观察到了线程A中操作“i=1”所产生的影响；现在，我们依然保持线程A和线程B之间的happen—before关系，同时线程C出现在了线程A和线程B的操作之间，但是C与B并没有happen—before关系，那么j的值就不确定了，线程C对变量i的影响可能会被线程B观察到，也可能不会，这时线程B就存在读取到不是最新数据的风险，不具备线程安全性。

    下面是Java内存模型中的八条可保证happen—before的规则，它们无需任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随机地重排序。

    1、程序次序规则：在一个单独的线程中，按照程序代码的执行流顺序，（时间上）先执行的操作happen—before（时间上）后执行的操作。

    2、管理锁定规则：一个unlock操作happen—before后面（时间上的先后顺序，下同）对同一个锁的lock操作。

    3、volatile变量规则：对一个volatile变量的写操作happen—before后面对该变量的读操作。

    4、线程启动规则：Thread对象的start（）方法happen—before此线程的每一个动作。

    5、线程终止规则：线程的所有操作都happen—before对此线程的终止检测，可以通过Thread.join（）方法结束、Thread.isAlive（）的返回值等手段检测到线程已经终止执行。

    6、线程中断规则：对线程interrupt（）方法的调用happen—before发生于被中断线程的代码检测到中断时事件的发生。

    7、对象终结规则：一个对象的初始化完成（构造函数执行结束）happen—before它的finalize（）方法的开始。

    8、传递性：如果操作A happen—before操作B，操作B happen—before操作C，那么可以得出A happen—before操作C。

##### 时间上先后顺序和happen—before原则

    ”时间上执行的先后顺序“与”happen—before“之间有何不同呢？

    1、**首先来看操作A在时间上先与操作B发生，是否意味着操作A happen—before操作B？**

    一个常用来分析的例子如下：

1. private int value = 0;
3. public int get(){
4. return value;
5. }
6. public void set(int value){
7. this.value = value;
8. }

private int value = 0;

public int get(){

return value;

}

public void set(int value){

this.value = value;

}

}

    假设存在线程A和线程B，线程A先（时间上的先）调用了setValue（3）操作，然后（时间上的后）线程B调用了同一对象的getValue（）方法，那么线程B得到的返回值一定是3吗？

    对照以上八条happen—before规则，发现没有一条规则适合于这里的value变量，从而我们可以判定线程A中的setValue（3）操作与线程B中的getValue（）操作不存在happen—before关系。因此，尽管线程A的setValue（3）在操作时间上先于操作B的getvalue（），但无法保证线程B的getValue（）操作一定观察到了线程A的setValue（3）操作所产生的结果，也即是getValue（）的返回值不一定为3（有可能是之前setValue所设置的值）。这里的操作不是线程安全的。

    因此，”一个操作时间上先发生于另一个操作“并不代表”一个操作happen—before另一个操作“。

    解决方法：可以将setValue（int）方法和getValue（）方法均定义为synchronized方法，也可以把value定义为volatile变量（value的修改并不依赖value的原值，符合volatile的使用场景），分别对应happen—before规则的第2和第3条。注意，只将setValue（int）方法和getvalue（）方法中的一个定义为synchronized方法是不行的，必须对同一个变量的所有读写同步，才能保证不读取到陈旧的数据，仅仅同步读或写是不够的 。

    2、**其次来看，操作A happen—before操作B，是否意味着操作A在时间上先与操作B发生？**

x = 1；

y = 2;

    假设同一个线程执行上面两个操作：操作A：x=1和操作B：y=2。根据happen—before规则的第1条，操作A happen—before 操作B，但是由于编译器的指令重排序（Java语言规范规定了JVM线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。这个过程通过叫做指令的重排序。指令重排序存在的意义在于：JVM能够根据处理器的特性（CPU的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能。在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整）等原因，操作A在时间上有可能后于操作B被处理器执行，但这并不影响happen—before原则的正确性。

    因此，”一个操作happen—before另一个操作“并不代表”一个操作时间上先发生于另一个操作“。

    最后，一个操作和另一个操作必定存在某个顺序，要么一个操作或者是先于或者是后于另一个操作，或者与两个操作同时发生。同时发生是完全可能存在的，特别是在多CPU的情况下。而两个操作之间却可能没有happen-before关系，也就是说有可能发生这样的情况，操作A不happen-before操作B，操作B也不happen-before操作A，用数学上的术语happen-before关系是个偏序关系。两个存在happen-before关系的操作不可能同时发生，一个操作A happen-before操作B，它们必定在时间上是完全错开的，这实际上也是同步的语义之一（独占访问）。

##### 利用happen—before规则分析DCL

   DCL即双重检查加锁，关于单例模式的DCL机制，可以参看：[**http://blog.csdn.net/ns\_code/article/details/17359719**](http://blog.csdn.net/ns_code/article/details/17359719)一文，这里不再详细介绍。下面是一个典型的在单例模式中使用DCL的例子：

1. public class LazySingleton {
2. private int someField;
4. private static LazySingleton instance;
6. private LazySingleton() {
7. this.someField = new Random().nextInt(200)+1;         // (1)
8. }
10. public static LazySingleton getInstance() {
11. if (instance == null) {                               // (2)
12. synchronized(LazySingleton.class) {               // (3)
13. if (instance == null) {                       // (4)
14. instance = new LazySingleton();           // (5)
15. }
16. }
17. }
18. return instance;                                      // (6)
19. }
21. public int getSomeField() {
22. return this.someField;                                // (7)
23. }
24. }

public class LazySingleton {

private int someField;

private static LazySingleton instance;

private LazySingleton() {

this.someField = new Random().nextInt(200)+1; // (1)

}

public static LazySingleton getInstance() {

if (instance == null) { // (2)

synchronized(LazySingleton.class) { // (3)

if (instance == null) { // (4)

instance = new LazySingleton(); // (5)

}

}

}

return instance; // (6)

}

public int getSomeField() {

return this.someField; // (7)

}

}

  这里得到单一的instance实例是没有问题的，问题的关键在于尽管得到了Singleton的正确引用，但是却有可能访问到其成员变量的不正确值。具体来说Singleton.getInstance().getSomeField()有可能返回someField的默认值0。如果程序行为正确的话，这应当是不可能发生的事，因为在构造函数里设置的someField的值不可能为0。为也说明这种情况理论上有可能发生，我们只需要说明语句(1)和语句(7)并不存在happen-before关系。

   假设线程Ⅰ是初次调用getInstance()方法，紧接着线程Ⅱ也调用了getInstance()方法和getSomeField()方法，我们要说明的是线程Ⅰ的语句(1)并不happen-before线程Ⅱ的语句(7)。线程Ⅱ在执行getInstance()方法的语句(2)时，由于对instance的访问并没有处于同步块中，因此线程Ⅱ可能观察到也可能观察不到线程Ⅰ在语句(5)时对instance的写入，也就是说instance的值可能为空也可能为非空。我们先假设instance的值非空，也就观察到了线程Ⅰ对instance的写入，这时线程Ⅱ就会执行语句(6)直接返回这个instance的值，然后对这个instance调用getSomeField()方法，该方法也是在没有任何同步情况被调用，因此整个线程Ⅱ的操作都是在没有同步的情况下调用 ，这时我们便无法利用上述8条happen-before规则得到线程Ⅰ的操作和线程Ⅱ的操作之间的任何有效的happen-before关系（主要考虑规则的第2条，但由于线程Ⅱ没有在进入synchronized块，因此不存在lock与unlock锁的问题），这说明线程Ⅰ的语句(1)和线程Ⅱ的语句(7)之间并不存在happen-before关系，这就意味着线程Ⅱ在执行语句(7)完全有**可能**观测不到线程Ⅰ在语句(1)处对someFiled写入的值，这就是DCL的问题所在。很荒谬，是吧？DCL原本是为了逃避同步，它达到了这个目的，也正是因为如此，它最终受到惩罚，这样的程序存在严重的bug，虽然这种bug被发现的概率绝对比中彩票的概率还要低得多，而且是转瞬即逝，更可怕的是，即使发生了你也不会想到是DCL所引起的。

    前面我们说了，线程Ⅱ在执行语句(2)时也有可能观察空值，如果是种情况，那么它需要进入同步块，并执行语句(4)。在语句(4)处线程Ⅱ还能够读到instance的空值吗？不可能。这里因为这时对instance的写和读都是发生在同一个锁确定的同步块中，这时读到的数据是最新的数据。为也加深印象，我再用happen-before规则分析一遍。线程Ⅱ在语句(3)处会执行一个lock操作，而线程Ⅰ在语句(5)后会执行一个unlock操作，这两个操作都是针对同一个锁--Singleton.class，因此根据第2条happen-before规则，线程Ⅰ的unlock操作happen-before线程Ⅱ的lock操作，再利用单线程规则，线程Ⅰ的语句(5) -> 线程Ⅰ的unlock操作，线程Ⅱ的lock操作 -> 线程Ⅱ的语句(4)，再根据传递规则，就有线程Ⅰ的语句(5) -> 线程Ⅱ的语句(4)，也就是说线程Ⅱ在执行语句(4)时能够观测到线程Ⅰ在语句(5)时对Singleton的写入值。接着对返回的instance调用getSomeField()方法时，我们也能得到线程Ⅰ的语句(1) -> 线程Ⅱ的语句(7)（由于线程Ⅱ有进入synchronized块，根据规则2可得），这表明这时getSomeField能够得到正确的值。但是仅仅是这种情况的正确性并不妨碍DCL的不正确性，一个程序的正确性必须在所有的情况下的行为都是正确的，而不能有时正确，有时不正确。

    对DCL的分析也告诉我们一条经验原则：对引用（包括对象引用和数组引用）的非同步访问，即使得到该引用的最新值，却并不能保证也能得到其成员变量（对数组而言就是每个数组元素）的最新值。

**解决方案：**

    1、最简单而且安全的解决方法是使用static内部类的思想，它利用的思想是：一个类直到被使用时才被初始化，而类初始化的过程是非并行的，这些都有JLS保证。

如下述代码：

1. public class Singleton {
3. private Singleton() {}
5. // Lazy initialization holder class idiom for static fields
6. private static class InstanceHolder {
7. private static final Singleton instance = new Singleton();
8. }
10. public static Singleton getSingleton() {
11. return InstanceHolder.instance;
12. }
13. }

public class Singleton {

private Singleton() {}

// Lazy initialization holder class idiom for static fields

private static class InstanceHolder {

private static final Singleton instance = new Singleton();

}

public static Singleton getSingleton() {

return InstanceHolder.instance;

}

}

#### 2、另外，可以将instance声明为volatile，即

**private** **volatile** **static** LazySingleton instance;

  这样我们便可以得到，线程Ⅰ的语句(5) -> 语线程Ⅱ的句(2)，根据单线程规则，线程Ⅰ的语句(1) -> 线程Ⅰ的语句(5)和语线程Ⅱ的句(2) -> 语线程Ⅱ的句(7)，再根据传递规则就有线程Ⅰ的语句(1) -> 语线程Ⅱ的句(7)，这表示线程Ⅱ能够观察到线程Ⅰ在语句(1)时对someFiled的写入值，程序能够得到正确的行为。

**注：**

**1、volatile屏蔽指令重排序的语义在JDK1.5中才被完全修复，此前的JDK中及时将变量声明为volatile，也仍然不能完全避免重排序所导致的问题（主要是volatile变量前后的代码仍然存在重排序问题），这点也是在JDK1.5之前的Java中无法安全使用DCL来实现单例模式的原因。**

**2、把volatile写和volatile读这两个操作综合起来看，在读线程B读一个volatile变量后，写线程A在写这个volatile变量之前，所有可见的共享变量的值都将立即变得对读线程B可见。**

    3、 在java5之前对final字段的同步语义和其它变量没有什么区别，在java5中，final变量一旦在构造函数中设置完成（前提是在构造函数中没有泄露this引用)，其它线程必定会看到在构造函数中设置的值。而DCL的问题正好在于看到对象的成员变量的默认值，因此我们可以将LazySingleton的someField变量设置成final，这样在java5中就能够正确运行了。

# 多线程并发知识点

#### Java对BIO、NIO、AIO的支持：

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。

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。

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

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方式使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。

#### IO和NIO的区别和原理？

（1）IO是面向流的，NIO是面向缓冲区的

Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它不能前后移动流中的数据。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

（2）阻塞IO与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

（3）选择器。使用单线程Selector来管理多个通道，减少系统开销，服务端和客户端各自维护一个管理通道的对象，我们称之为selector，该对象能检测一个或多个通道 (channel) 上的事件。

Socket默认的是1024×8=8192字节

JAVA IO流的程序一般分以下四步：

1.面向字符的输入流类：Reader >> InputStreamReader >> FileReader

从文件读取的输入流: FileReader fr = new FileReader(new File("xxx.txt"));//FileReader没有实现父类中带字符集参数的构造函数，只能按系统默认的字符集来解码

将字节转换到字符的输入流: InputStreamReader isr = new InpuStreamReader(new FileInputStream("xxx.txt")， "UTF-8"));

2.如果用缓冲区提高读写性能 BufferReader br = new BufferReader(fr);

3.开始读写操作 while((s=br.readLine()!=null) { sb.append(s); }

Java NIO:

Channel :  可以和原来的Stream类比， 但是有个关键区别， 那就是通过Channel 读写数据，是非阻塞的， 一个socket 也是Channel 的一种。

Buffer :  通过Channel 读写的数据都在Buffer 中， 由于Buffer 不是流， 你读到Buffer 尾部以后还可以从头再读。

Selector ：  和Channel配合使用， Channel 可以把自己注册到Selector当中， 告诉Selector 说， 我要监听XXX事件， 这是一个线程管理多个Channel的关键。

#### ThreadLocal类

为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。

它主要由四个方法组成initialValue()，get()，set(T)，remove()，其中值得注意的是initialValue()，该方法是一个protected的方法，显然是为了子类重写而特意实现的。

ThreadLocal的原理:在ThreadLocal类中有一个Map，用于存储每一个线程的变量的副本。

private Map values = Collections.synchronizedMap(new HashMap());

Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离

#### 线程中sleep与wait的区别？

sleep()方法-interrupt()

属于Thread类静态方法；不释放同步锁（占着cpu资源）；谁调用谁暂停（当前线程）；可在任何地方使用；必须捕获异常；sleep可被interrupt()强行打断，sleep时间到后，可恢复执行

wait()方法-notify()

属于Object类；释放同步锁（释放cpu资源）；由某个确定的对象调用（当前对象）；wait、notify、notifyAll只能用在synchronize块中；不必捕获异常；wait可被notify()唤起，时间到后，就绪队列。

#### 当前线程怎么让它“暂停”，等结果回来后，再向后执行？

答：先生成一个对象obj，在一个全局map里put(ID，obj)存放起来，再用synchronized获取obj锁，再调用obj.wait()让当前线程处于等待状态，然后另一消息监听线程等到服务端结果来了后，再map.get(ID)找到obj，再用synchronized获取obj锁，再调用obj.notifyAll()唤醒前面处于等待状态的线程。

thread.Join把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。

t.join();      //使调用线程 t 在此之前执行完毕。

t.join(1000);  //等待 t 线程，等待时间是1000毫秒

进程三种基本状态：

1、就绪（Ready）状态

当进程已分配到除CPU以外的所有必要资源后，只要在获得CPU，便可立即执行，进程这时的状态就称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将他们排成一个队列，称为就绪队列。

2、执行状态

进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态；再多处理机系统中，则有多个进程处于执行状态。

3、阻塞状态

正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，亦即程序的执行受到阻塞，把这种暂停状态称为阻塞状态，有时也称为等待状态或封锁状态。

#### NIO中select的实现机制

(1)  创建Selector

(2)  向Selector注册通道

(3)  SelectionKey

(4)  通过Selector选择通道

(5)  wakeup

(6)  close()

ServerSocketChannel编程操作：

    // 创建选择器

    Selector selector=Selector.open();

    // 打开监听信道

    ServerSocketChannel listenerChannel=ServerSocketChannel.open();

    // 与本地端口绑定

    listenerChannel.socket().bind(new InetSocketAddress(ListenPort));

    // 设置为非阻塞模式

    listenerChannel.configureBlocking(false);

    // 将选择器绑定到监听信道，只有非阻塞信道才可以注册选择器。并在注册过程中指出该信道可以进行Accept操作

    listenerChannel.register(selector， SelectionKey.OP\_ACCEPT);

    // 创建一个处理协议的实现类，由它来具体操作

    TCPProtocol protocol=new TCPProtocolImpl(BufferSize);

    // 反复循环，等待IO

    while(true){

      }

#### dns解析过程

1、浏览器中输入[www.qq.com域名，先检查本地的hosts](http://www.qq.com域名，先检查本地的hosts)文件映射关系，若有，就调用这个IP地址映射，完成域名解析。

2、若hosts文件无，则查找本地DNS解析器缓存，若有，直接返回域名解析。

3、若本地DNS解析器缓存无，首先会找TCP/IP参数中设置的首选DNS服务器(本地DNS服务器)，若有，直接返回域名解析。

4、若本地DNS服务器无，但该服务器缓存了此网址映射关系或设置了转发器，则返回映射或转发至上一级查询。

5、若无，本地DNS就把请求发至13台根DNS。

根DNS查找：根DNS->顶级域名服务器IP->.com域的下一级DNS服务器地址(qq.com)->qq.com域服务器->[www.qq.com主机](http://www.qq.com主机)。

-------------------------------------------------------------

Mina

基于java nio类库开发，快速开发应用；

采用非阻塞方式异步传输，事件驱动；

支持tcp、udp协议，支持批量数据传输；

灵活的过滤器加载机制，网络i/o编码，消息编解码；

松耦合，控制反转的设计模式，业务逻辑互相分离

Mina框架机制

（1）IoService处理IO操作

IoAccepter:NioSocketAccepter

IoConnector:NioSocketConnector

（2）IoProcessor线程处理

主控线程，检查SocketChannel中数据，并调用IoFilter处理连接请求

（3）IoFilter过滤器链，日志处理、字节变换、对象转换等操作

AOP模式，事件，日志，数据转换codec

ProtocolCodecFactor：Encoder、Decoder

（4）IoHandler：IoHandleAdapter真正的处理业务逻辑的地方

exceptionCaught()捕获异常

messageReceived()消息到达

messageSent()发送消息

sessionClosed()连接关闭

sessionCreated()连接创建

sessionIdle()连接空闲

sessionOpened()连接打开

（5）IoSession保存会话属性和发送信息

Mina编码操作：

IoAcceptor acceptor = new NioSocketAcceptor();

        acceptor.getFilterChain().addLast( "logger"， new LoggingFilter() );

        acceptor.getFilterChain().addLast( "codec"， new ProtocolCodecFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));

        acceptor.setHandler( new TimeServerHandler() );

    acceptor.getSessionConfig().setReadBufferSize( 2048 );

        acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH\_IDLE， 10 );

        acceptor.bind( new InetSocketAddress(PORT) );

装饰器模式：装饰器模式的本质就是动态组合。

    (1)需要在不影响其他对象的情况下，以动态、透明的方式给对象添加职责。

    (2)如果不适合使用子类来进行扩展的时候，可以考虑使用装饰器模式。

Java中线程的创建有两种方式：

1．  通过继承Thread类，重写Thread的run()方法，将线程运行的逻辑放在其中

2．  通过实现Runnable接口，实例化Thread类

public static void main(String[] args) {

//implements Runnable

MyThread myThread = new MyThread();

Thread t1 = new Thread (myThread，"1");

Thread t2 = new Thread (myThread，"2");

t1.start();

t2.start();

//extends Thread

MyThread2 tt1 = new MyThread2();

MyThread2 tt2 = new MyThread2();

tt1.start();

tt2.start();

}

#### Thread和Runnable的区别和联系

在程序开发中只要是多线程肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下好处：

1.避免点继承的局限，一个类可以继承多个接口。

2.适合于资源的共享

3.Thread类是Runnable接口的子类（public class Thread extends Object implements Runnable）

#### 多次start一个线程会怎么样

java.lang.IllegalThreadStateException

一个线程对象只能调用一次start方法。从new到等待运行是单行道，所以如果你对一个已经启动的线程对象再调用一次start方法的话，会产生:IllegalThreadStateException异常。

线程有哪些状态

大体说一下线程的5种状态：新建、就绪、运行、阻塞、死亡。

新建状态（new）： 线程刚刚创建，最初始的状态。

就绪状态（runnable）： 线程创建完成后，并且运行的时机已经成熟，确定要执行了，但是还没执行。（没有分配到cpu）

运行状态（running）： 线程真真正正的获得了cpu，然后执行相应的操作。（带啊跑起来了）

阻塞状态（blocked）： 线程跑到一半，由于外界发生的变化，在运行到某个地方然后停住不运行了。（由于外界或自己的原因，交出了cpu，但是和就绪不同，这个时候线程可以运行的条件并不满足，所以连就绪都不是，而是一直挂起没有任何操作。）

死亡状态（dead）： 线程已经完成了自己的使命，真正的结束了，并且不再会有该线程。

假如有Thread1、Thread2、Thread3、Thread4四条线程分别统计C、D、E、F四个盘的大小，所有线程都统计完毕交给Thread5线程去做汇总，应当如何实现？

java.util.concurrent下就有现成的类可以使用

#### 9.HTTP长连接和短连接的区别

TCP/IP

TCP/IP是个协议组，可分为三个层次：网络层、传输层和应用层。

在网络层有IP协议、ICMP协议、ARP协议、RARP协议和BOOTP协议。

在传输层中有TCP协议与UDP协议。

在应用层有:TCP包括FTP、HTTP、TELNET、SMTP等协议

           UDP包括DNS、TFTP等协议

短连接

连接->传输数据->关闭连接

HTTP是无状态的，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。

也可以这样说：短连接是指SOCKET连接后发送后接收完数据后马上断开连接。

长连接

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

http的长连接

HTTP也可以建立长连接的，使用Connection:keep-alive，HTTP 1。1默认进行持久连接.HTTP1.1和HTTP1.0相比较而言，最大的区别就是增加了持久连接支持(貌似最新的 http1.0 可以显示的指定 keep-alive)，但还是无状态的，或者说是不可以信任的。

#### 什么时候用长连接，短连接？

 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

总之，长连接和短连接的选择要视情况而定。

# 分布式网站系统

#### 大型网站系统的特点

高并发，大流量；高可用；海量数据；用户分布广泛，网络情况复杂；安全环境恶劣；需求快速变更，发布频繁；渐进式发展

#### 大型网站架构演化

(1)初始阶段：独立服务器

应用服务器、文件服务、数据库服务所有资源在同一台服务器

(2)起步阶段：应用与数据分离，不同特性服务器承担不同服务角色。

应用服务器，文件服务器和数据库服务器对硬件资源要求各不相同，分离承担不同角色服务

(3)提升阶段：使用缓存技术和数据库读写分离

网站访问遵循二八定律，缓存热点数据，提升网站访问速度，降低数据服务访问压力

(4)进化阶段1：应用服务器集群：实现负载均衡

集群是解决高并发，海量数据问题的常用手段，引入负载均衡服务器实现系统访问请求的分发，实现系统的可伸缩性

(5)进化阶段2：分布式数据库，实现读写分离

用户达到一定规模，缓存仍不能解决数据库服务瓶颈问题，搭建数据库主从架构，实现读写分离

(6)进化阶段3：面对复杂网络环境，使用反向代理和CDN加速网站响应，提升缓存能力，加快用户访问速度

反向代理：部署在网站中心机房；CDN(内容分发网络)：部署在网络提供商机房

#### 大型网站架构模式

(1)分层：横向切分：应用层，服务层，数据层

分层将系统在横向维度上切分，通过上层对下层的依赖和调用组成完整系统，每层职责单一。

(2)分割：纵向切分：分子系统、分模块

在分层的基础上，将同一层按照功能和服务将系统分割包装成高内聚低耦合的模块单元

(3)分布式：分层和分割的目的都是为了分布式部署，分布式应用服务，分布式静态资源，分布式数据存储，分布式计算

(4)集群化：部署相同应用构成集群，通过负载均衡提供服务

(5)缓存：改善系统性能第一手段，使用前提条件，数据访问热点不均衡，数据在某段时间内有效，不会很快过期

(6)异步：引入消息队列，实现应用异步解耦，削峰填谷作用

(7)冗余：搭建灾备数据中心，冗余实现高可用

#### 大型网站架构要素

(1)性能：

web前端：减少http请求，浏览器缓存，启用页面压缩，合理布局页面(CSS最上，JS最下)，减少cookie传输

缓存方面：使用反向代理和CDN，缓存热点文件

应用服务器端：异步操作至消息队列，服务器本地缓存，分布式缓存

代码层面：多线程，资源复用，数据结构，垃圾回收，优化内存管理

数据库端：索引，缓存，SQL优化

(2)可用性

高可用的手段是冗余，建立灾备中心，实现99.99%可用性

(3)伸缩性

伸缩性指通过不断向集群中加入服务器即可缓解不断上升的并发压力和数据需求

(4)扩展性

扩展性架构直接关系系统的功能性需求，主要手段是事件驱动架构（消息队列）和分布式服务（业务和可复用服务分离）

(5)安全性

系统性能就是两个事：

Throughput ，吞吐量。也就是每秒钟可以处理的请求数，任务数。

Latency， 系统延迟。也就是系统在处理一个请求或一个任务时的延迟。

系统类型：CPU 密集型、I/O 密集型

#### 分布式概念

(1)三元组(节点+网络+存储)：

分布式系统说白了，就是很多机器组成的集群，靠彼此之间的网络通信，担当的角色可能不同，共同完成同一个事情的系统。

1、节点 -- 系统中按照协议完成计算工作的一个逻辑实体，可能是执行某些工作的进程或机器

2、网络 -- 系统的数据传输通道，用来彼此通信。通信是具有方向性的。

3、存储 -- 系统中持久化数据的数据库或者文件存储。

(2)状态特性

各个节点的状态可以是“无状态”或者“有状态的”.

一般认为，节点是偏计算和通信的模块，一般是无状态的。这类应用一般不会存储自己的中间状态信息，比如Nginx，一般情况下是转发请求而已，不会存储中间信息。另一种“有状态”的，如MySQL等数据库，状态和数据全部持久化到磁盘等介质。“无状态”的节点一般我们认为是可随意重启的，因为重启后只需要立刻工作就好。“有状态”的则不同，需要先读取持久化的数据，才能开始服务。所以，“无状态”的节点一般是可以随意扩展的，“有状态”的节点需要一些控制协议来保证扩展。

(3)系统异常

异常，可认为是节点因为某种原因不能工作，此为节点异常。还有因为网络原因，临时、永久不能被其他节点所访问，此为网络异常。在分布式系统中，要有对异常的处理，保证集群的正常工作。

#### 分布式系统与单节点的不同

  1、网络传输

在unix/linux/mac(类Unix)环境下，两个机器通信（socket），传输数据是read()/write()系统调用，把一段内存缓冲区发出去。发送数据需要走内核->网卡->链路->对端网卡->内核，路径太长只能是异步操作。write()把数据写入内核缓冲区之后就返回到应用层了，具体后面何时发送、怎么发送、TCP怎么做滑动窗口、流控都是tcp/ip协议栈内核的事情了。

所以在应用层，能确认对方受到了消息只能是对方应用返回数据，逻辑确认了这次发送才认为是成功的。

这就却别与单系统编程，大部分系统调用、库调用只要返回了就说明已经确认完成了。

  2、不可控的状态

在单系统编程中，我们对系统状态是非常可控的。比如函数调用、逻辑运算，要么成功，要么失败，因为这些操作被框在一个机器内部，cpu/总线/内存都是可以快速得到反馈的。开发者可以针对这两个状态很明确的做出程序上的判断和后续的操作。

而在分布式的网络环境下，这就变得微妙了。比如一次rpc、http调用，可能成功、失败，还有可能是“超时”，这就比前者的状态多了一个不可控因素，导致后面的代码不是很容易做出判断。试想一下，用A用支付宝向B转了一大笔钱，当他按下“确认”后，界面上有个圈在转啊转，然后显示请求超时了，然后A就抓狂了，不知道到底钱转没转过去，开始确认自己的账户、确认B的账户、打电话找客服等等。

所以分布式环境下，我们的其实要时时刻刻考虑面对这种不可控的“第三状态”设计开发，这也是挑战之一。

  3、视”异常“为”正常“

单系统下，进程/机器的异常概率十分小。即使出现了问题，可以通过人工干预重启、迁移等手段恢复。

但在分布式环境下，机器上千台，每几分钟都可能出现宕机、死机、网络断网等异常，出现的概率很大。所以，这种环境下，进程core掉、机器挂掉都是需要我们在编程中认为随时可能出现的，这样才能使我们整个系统健壮起来，所以”容错“是基本需求。

异常可以分为如下几类：

节点错误：一般是由于应用导致，一些coredump和系统错误触发，一般重新服务后可恢复。

硬件错误：由于磁盘或者内存等硬件设备导致某节点不能服务，需要人工干预恢复。

网络错误：由于点对点的网络抖动，暂时的访问错误，一般拓扑稳定后或流量减小可以恢复。

网络分化：网络中路由器、交换机错误导致网络不可达，但是网络两边都正常，这类错误比较难恢复，并且需要在开发时特别处理。【这种情况也会比较前面的问题较难处理】

#### 分布式系统设计策略

  1、重试机制

一般情况下，写一段网络交互的代码，发起rpc或者http，都会遇到请求超时而失败情况。可能是网络抖动(暂时的网络变更导致包不可达，比如拓扑变更)或者对端挂掉。这时一般处理逻辑是将请求包在一个重试循环块里，如下：

int retry = 3;

while(!request() && retry--)

  sched\_yield();   // or usleep(100)

此种模式可以防止网络暂时的抖动，一般停顿时间很短，并重试多次后，请求成功！但不能防止对端长时间不能连接(网络问题或进程问题)

  2、心跳机制

心跳顾名思义，就是以固定的频率向其他节点汇报当前节点状态的方式。收到心跳，一般可以认为一个节点和现在的网络拓扑是良好的。当然，心跳汇报时，一般也会携带一些附加的状态、元数据信息，以便管理。但心跳不是万能的，收到心跳可以确认ok，但是收不到心跳却不能确认节点不存在或者挂掉了，因为可能是网络原因倒是链路不通但是节点依旧在工作。

所以切记，”心跳“只能告诉你正常的状态是ok，它不能发现节点是否真的死亡，有可能还在继续服务。(后面会介绍一种可靠的方式 -- Lease机制)

  3、副本

副本指的是针对一份数据的多份冗余拷贝，在不同的节点上持久化同一份数据，当某一个节点的数据丢失时，可以从副本上获取数据。数据副本是分布式系统解决数据丢失异常的仅有的唯一途径。当然对多份副本的写入会带来一致性和可用性的问题，比如规定副本数为3，同步写3份，会带来3次IO的性能问题。还是同步写1份，然后异步写2份，会带来一致性问题，比如后面2份未写成功其他模块就去读了(下个小结会详细讨论如果在副本一致性中间做取舍)。

  4、中心化/无中心化

系统模型这方面，无非就是两种：

中心节点，例如mysql的MSS单主双从、MongDB Master、HDFS NameNode、MapReduce JobTracker等，有1个或几个节点充当整个系统的核心元数据及节点管理工作，其他节点都和中心节点交互。这种方式的好处显而易见，数据和管理高度统一集中在一个地方，容易聚合，就像领导者一样，其他人都服从就好。简单可行。

但是缺点是模块高度集中，容易形成性能瓶颈，并且如果出现异常，就像群龙无首一样。

无中心化的设计，例如cassandra、zookeeper，系统中不存在一个领导者，节点彼此通信并且彼此合作完成任务。好处在于如果出现异常，不会影响整体系统，局部不可用。缺点是比较协议复杂，而且需要各个节点间同步信息。

#### 缓存设计 基本原理：

create/update/delete---同时存到数据库和缓存系统

query--先从缓存系统查，没有记录才从数据库查，并把从数据库查的结果也放一份到缓存系统

(1)单机单线程版：

Map<String, T> cache = new HashMap<String, T>();

增改-->put

删-->remove

查-->get

(2)单机多线程版：防止对数据造成破坏

ConcurrentHashMap<String, T> cache = new ConcurrentHashMap<String, T>();

(3)分布式版:

a.master-slave型:保证数据一致性

节点通信存在两种模型：共享内存(Shared memory)和消息传递(Messages passing)

Paxos算法就是一种基于消息传递模型的一致性算法。

b.masterA1-slaveA2+masterB1-slaveB2型:圆环式的一致性Hash算法，计算数据的存放地址

#### 分布式系统概念：

分布式系统是由一系列分散自治组件通过互联网并行并发协作，从而组成的一个coherent软件系统。

它具备资源共享，并行并发，可靠容错，透明开放等特性。

#### 分布式基础理论：CAP，BASE，Paxos，事务。

（1）CAP理论:一致性（Consistency），可用性（Availability），分区容忍性（Partition tolerance）。

1.Consistency一致性，Eric Brewer（CAP理论提出者）用一个服务要么被执行，要么不被执行来定义(原文：A service that is consistent operates fully or not at all)。请注意，这里的一致性是有别于数据库ACID属性中的C，数据库层面的C指的是数据的操作不能破坏数据之间的完整性约束，如外键约束。在分布式环境中，可以把C简单理解为多节点看到的是数据单一或者同一副本。

2.Availability可用性，意味着服务是可用的。可用性又可以细分为写可用和读可用。在分布式环境中，往往指的是系统在确定时间内可返回读写操作结果，也即读写均可用。

3.Partition tolerance分区容忍性，除了整个网络故障外（如光纤被掘断），其它故障（如丢包、乱序、抖动、甚至是网络分区节点 crash ）都不能导致整个系统无法正确响应。

（2）BASE理论:基本可用（Basically Available）、软状态（ Soft State）、最终一致性（ Eventual Consistency）

eBay的架构师Dan Pritchett，在ACM上发表文章提出。BASE理论是对CAP理论的延伸，核心思想是即使无法做到强一致性（Strong Consistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。

1.基本可用（Basically Available）是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。

电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务。这就是损失部分可用性的体现。

2.软状态（ Soft State）软状态是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。mysql replication的异步复制也是一种体现。

3.最终一致性（ Eventual Consistency）是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况。

（3）Paxos:分布式一致性算法

提议者Proposer、接受者Acceptor、提案Proposal的proposal\_id

总体说来，paxos就是通过两个阶段确定一个决议：

Phase1：确定谁的编号proposal\_id最高，只有编号最高者才有权利提交proposal；

Phase2：编号最高者提交proposal，如果没有其他节点提出更高编号的proposal，则该提案会被顺利通过；否则，整个过程就会重来。

你编号高，我比你更高，反复如此，算法永远无法结束，这叫活锁。活锁便是Paxos无法解决的硬伤.

（4）事务:

分布式事务，常见两个处理办法就是两段式提交和补偿。

1.两段式提交:

a. 协调员服务器发送一条投票请求消息给所有参与这次事务的参与者服务器。

b. 当一个参与者收到一条投票请求，它会向协调员发送一条响应请求消息:YES/NO。如果参与者投票NO，意味着参与者不参与这次事务，等价于ABORT决定，事务终止。

c. 协调员收集所有参与者的投票都是YES，则COMMIT，并把COMMIT消息发送给所有参与者。否则ABORT，协调员会把ABORT消息发给那些投票为YES的那些参与者

2.补偿

先处理业务，然后定时或者回调里，检查状态是不是一致的，如果不一致采用某个策略，强制状态到某个结束状态。

思路：事务拆分，大的业务流程，转化成几个小的业务流程，加入中间状态，考虑最终一致性。

复杂的业务交互过程中，不建议使用强一致性的分布式事务。解决分布式事务的最好办法就是不考虑分布式事务。就像刚说的问题一样，把分布式的事务过程拆解成多个中间状态，中间状态的东西不允许用户直接操作，等状态都一致成功，或者检测到不一致的时候全部失败掉。就解耦了这个强一致性的过程。

---

分布式要点：

节点之间使用JMS、RMI或RESTful方式通信等

---

接口防重入，幂等性：

函数/接口可以使用相同的参数重复执行, 不应该影响系统状态, 也不会对系统造成改变 .

幂等接口的内部实现需要有对内保护机制, 一般情况是用类似于乐观锁的版本机制.版本重点是体现时间的先后.

引入 票据 (token) ：

从接口实现上：

请求接口时，生成唯一的token，并记录到db，执行非幂等操作时必须先判断该token是否已执行过。

从业务规划上：

比如当用户点击赞同时，将答案的赞同数+1。改为当用户点击赞同时，确保答案赞同表中存在一条记录(唯一性约束)，用户、答案。

---

分布式用到的一些技术：

序列化技术：

Serializable，序列化是将对象状态转换为可保持或传输的形式的过程。 序列化的补集是反序列化，后者将流转换为对象。 这两个过程一起保证数据易于存储和传输。

实现分布式的时候，需要将一个对象从Server分发给client。通过网络TCP，建立Socket，传输一个对象，就需要将对象转换成一段字节流，即对象的序列化。同时，也要求可以从这段字节流，创建出对应的对象出来。

---

java Serializable存储的文件内容：

当前类描述

当前类属性描述

超类描述

超类属性描述(如果超类还有超类，则依次递归)

超类属性值描述

子类属性值描述

---

序列化框架

（1）Java Serializable

java序列化主要是为了跨平台，实现对象的一致性，可在不同的平台上，保持自己原有的属性和方法不变

Serializable方法： writeObject()  readObject()

缺点：

1.无法跨语言。

2.序列后的码流太大。java序列化的大小是二进制编码的5倍多。

3.序列化性能太低。java序列化的性能只有二进制编码的6.17倍。

（2）Kryo是一个快速高效的Java序列化框架，旨在提供快速、高效和易用的API。无论文件、数据库或网络数据Kryo都可以随时完成序列化。Kryo还可以执行自动深拷贝（克隆）、浅拷贝（克隆）。这是对象到对象的直接拷贝，非对象->字节->对象的拷贝。

（3）Protobuf是google开源的项目，全称 Google Protocol Buffers.特点：

1.结构化数据存储格式（xml,json等）

2.高性能编解码技术

3.语言和平台无关，扩展性好

4.支持java,C++,Python三种语言。

（4）Thrift源于faceBook，2007年facebook将Thrift做为一个开源项目交给了apache基金会。特点：

1.Thrift支持多种语言（C++,C#,Cocoa,Erlag,Haskell,java,Ocami,Perl,PHP,Python,Ruby,和SmallTalk）

2.Thrift适用了组建大型数据交换及存储工具，对于大型系统中的内部数据传输，相对于Json和xml在性能上和传输大小上都有明显的优势。

3.Thrift支持三种比较典型的编码方式。（通用二进制编码，压缩二进制编码，优化的可选字段压缩编解码）

（5）Hessian

Hessian 是由 caucho 提供的一个基于 binary-RPC 实现的远程通讯 library

4种Cache实现

1LRU，最后使用的排到前面。Cache溢出时，最远被使用的就被clear。

2FIFO，先进先出。

3Memory，内存引用。该实现无数量限制。前两种是基于jvm实现。

WEAK，产生内存回收动作时，失效。

SOFT，内存不足时，失效。

STRONG，显式刷新时，失效。

4OsCache(支持分布式)。通过oscahce.properties控制，是通过第三方的缓存插件实现。

适应范围

频繁查询，很少更改的内容。如：分类等。

1+n查询。n是父类，数据量较少。如：查询Spu信息时，同意需要获得其品类信息。

效率低，执行频率高的SQL。如统计一类的SQL。

有了Cache机制后，1+n不再可怕。

---

被缓存的数据一般具有以下特点：

1、经常被访问   不经常被访问的数据，即使缓存了对系统的性能吞吐也没太大的改善，没什么必要做缓存，直接访问DB、File、其他系统即可

2、改动不频繁  如果一个数据改动很频繁，缓存的数据很容易就过期或者失效，保证数据一致性成本很高，命中率也很难上去，缓存的效果也不会好

3、时效性不强  业务上要能容忍缓存失效前数据的不准确性

---

为哪些数据做缓存？

模型对象，这在业务逻辑层面最常见。

数据库查询结果集。

页面缓存、页面片段缓存。

运算结果集，尤其对于幂等性服务。

外部接口查询结果。

---

缓存系统的核心模型应该包括哪些？

CacheManager：模型管理对象，可以是多实例的，也可以是单实例的。

Cache：通过CacheManager创建出来的缓存容器，内部包含了真正的缓存承载体，至少开放add/remove/flush等接口。

CacheMap：真正的缓存承载体，大致上都是一个Map，各种类型的Map。

CacheEntity：缓存条目，相当于CacheMap里面的每一条Entry。

CacheEvent：缓存事件，比如CacheEntity的创建、更新、删除等等。

CacheEventListener：缓存事件相应的监听器。

CacheEvictionAlgorithm：缓存淘汰算法，常见的有LRU、LFU、FIFO等等。

---

从请求和数据流向的角度看，一个完整的缓存框架应该包括这样几个部分：

操作捕获：Cache

缓存数据存储：put(CacheEntity)

缓存数据读取：get(CacheEntity)

缓存数据流动：remove(CacheEntity)

---

以缓存数据存储为例，可以拆分成：

key生成、value封装、元数据封装

索引生成、文件结构生成

序列化、反序列化

淘汰算法、过期检查

存储数据预处理、持久化媒介

---

缓存数据的淘汰

1. 给缓存加tag，2. 版本号（必须单调递增，时间戳是最好的选择）3. 提供手动清理缓存的接口。

readOnly，表示Cache对象是否只读。False，表示外部更改cache内容无效。

Serialize，是否序列化。true，表示存贮到cache中的是系列化后的对象。

组合：

readOnly=false, Serialize=false：Cache Session有效。如:1+n时，下次1+n将会失效。 不系列化，外部更改有效。

readOnly=true, Serialize=false：所有session共享Cache，取出时实例是同一个。不系列化，外部更改有效。默认的

readOnly=false, Serialize=true：所有session共享Cache，取出时实例不一样，但是内容一样。 系列化，外部更改无效

readOnly=true, Serialize=true： 同默认效果一样。

看得出，主要是通过系列化来保证外部更改属性后不影响其它session的取出的结果。

---

一致性Hash算法

一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：

1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去。

3、分散性(Spread)：(即不一致)在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。

4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

# Kafka

Kafka是分布式发布-订阅消息系统。它最初由LinkedIn公司开发，之后成为Apache项目的一部分。Kafka是一个分布式的，可划分的，冗余备份的持久性的日志服务。它主要用于处理活跃的流式数据。

  在[**大数据**](http://lib.csdn.net/base/hadoop)系统中，常常会碰到一个问题，整个大数据是由各个子系统组成，数据需要在各个子系统中高性能，低延迟的不停流转。传统的企业消息系统并不是非常适合大规模的数据处理。为了能在同时搞定在线应用（消息）和离线应用（数据文件，日志）Kafka就出现了。

Kafka可以起到两个作用：

1. 降低系统组网复杂度。
2. 降低编程复杂度，各个子系统不在是相互协商接口，各个子系统类似插口插在插座上，Kafka承担高速数据总线的作用。

#### Kafka主要特点：

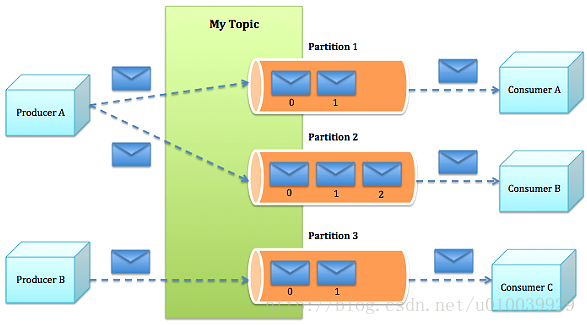
1. 同时为发布和订阅提供高吞吐量。据了解，Kafka每秒可以生产约25万消息（50 MB），每秒处理55万消息（110 MB）。
2. 可进行持久化操作。将消息持久化到磁盘，因此可用于批量消费，例如ETL，以及实时应用程序。通过将数据持久化到硬盘以及replication防止数据丢失。
3. 分布式系统，易于向外扩展。所有的producer、broker和consumer都会有多个，均为分布式的。无需停机即可扩展机器。
4. 消息被处理的状态是在consumer端维护，而不是由server端维护。当失败时能自动平衡。
5. 支持online和offline的场景。   
   Kafka的架构：



Kafka的整体[**架构**](http://lib.csdn.net/base/architecture)非常简单，是显式分布式架构，producer、broker（kafka）和consumer都可以有多个。Producer，consumer实现Kafka注册的接口，数据从producer发送到broker，broker承担一个中间缓存和分发的作用。broker分发注册到系统中的consumer。broker的作用类似于缓存，即活跃的数据和离线处理系统之间的缓存。客户端和服务器端的通信，是基于简单，高性能，且与编程语言无关的TCP协议。几个基本概念：

1. Topic：特指Kafka处理的消息源（feeds of messages）的不同分类。
2. Partition：Topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列。partition中的每条消息都会被分配一个有序的id（offset）。
3. Message：消息，是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息。
4. Producers：消息和数据生产者，向Kafka的一个topic发布消息的过程叫做producers。
5. Consumers：消息和数据消费者，订阅topics并处理其发布的消息的过程叫做consumers。
6. Broker：缓存代理，Kafka集群中的一台或多台服务器统称为broker。

#### 消息发送的流程：



1. Producer根据指定的partition方法（round-robin、hash等），将消息发布到指定topic的partition里面
2. kafka集群接收到Producer发过来的消息后，将其持久化到硬盘，并保留消息指定时长（可配置），而不关注消息是否被消费。
3. Consumer从kafka集群pull数据，并控制获取消息的offset

#### Kafka的设计

**1、吞吐量**

  高吞吐是kafka需要实现的核心目标之一，为此kafka做了以下一些设计：

1. 数据磁盘持久化：消息不在内存中cache，直接写入到磁盘，充分利用磁盘的顺序读写性能
2. zero-copy：减少IO操作步骤
3. 数据批量发送
4. 数据压缩
5. Topic划分为多个partition，提高parallelism

**2、负载均衡**

1. producer根据用户指定的算法，将消息发送到指定的partition
2. 存在多个partiiton，每个partition有自己的replica，每个replica分布在不同的Broker节点上
3. 多个partition需要选取出lead partition，lead partition负责读写，并由zookeeper负责fail over
4. 通过zookeeper管理broker与consumer的动态加入与离开

**3、拉取系统**

由于kafka broker会持久化数据，broker没有内存压力，因此，consumer非常适合采取pull的方式消费数据，具有以下几点好处：

1. 简化kafka设计
2. consumer根据消费能力自主控制消息拉取速度
3. consumer根据自身情况自主选择消费模式，例如批量，重复消费，从尾端开始消费等

**4、可扩展性**

当需要增加broker结点时，新增的broker会向zookeeper注册，而producer及consumer会根据注册在zookeeper上的watcher感知这些变化，并及时作出调整。

#### Kafka的应用场景：

1、消息队列

比起大多数的消息系统来说，Kafka有更好的吞吐量，内置的分区，冗余及容错性，这让Kafka成为了一个很好的大规模消息处理应用的解决方案。消息系统一般吞吐量相对较低，但是需要更小的端到端延时，并尝尝依赖于Kafka提供的强大的持久性保障。在这个领域，Kafka足以媲美传统消息系统，如ActiveMR或RabbitMQ。

2、行为跟踪

Kafka的另一个应用场景是跟踪用户浏览页面、搜索及其他行为，以发布-订阅的模式实时记录到对应的topic里。那么这些结果被订阅者拿到后，就可以做进一步的实时处理，或实时监控，或放到[**Hadoop**](http://lib.csdn.net/base/hadoop)/离线数据仓库里处理。

3、元信息监控

作为操作记录的监控模块来使用，即汇集记录一些操作信息，可以理解为运维性质的数据监控吧。

4、日志收集

日志收集方面，其实开源产品有很多，包括Scribe、Apache Flume。很多人使用Kafka代替日志聚合（log aggregation）。日志聚合一般来说是从服务器上收集日志文件，然后放到一个集中的位置（文件服务器或HDFS）进行处理。然而Kafka忽略掉文件的细节，将其更清晰地抽象成一个个日志或事件的消息流。这就让Kafka处理过程延迟更低，更容易支持多数据源和分布式数据处理。比起以日志为中心的系统比如Scribe或者Flume来说，Kafka提供同样高效的性能和因为复制导致的更高的耐用性保证，以及更低的端到端延迟。

5、流处理

这个场景可能比较多，也很好理解。保存收集流数据，以提供之后对接的Storm或其他流式计算框架进行处理。很多用户会将那些从原始topic来的数据进行阶段性处理，汇总，扩充或者以其他的方式转换到新的topic下再继续后面的处理。例如一个文章推荐的处理流程，可能是先从RSS数据源中抓取文章的内容，然后将其丢入一个叫做“文章”的topic中；后续操作可能是需要对这个内容进行清理，比如回复正常数据或者删除重复数据，最后再将内容匹配的结果返还给用户。这就在一个独立的topic之外，产生了一系列的实时数据处理的流程。Strom和Samza是非常著名的实现这种类型数据转换的框架。

6、事件源

事件源是一种应用程序设计的方式，该方式的状态转移被记录为按时间顺序排序的记录序列。Kafka可以存储大量的日志数据，这使得它成为一个对这种方式的应用来说绝佳的后台。比如动态汇总（News feed）。

7、持久性日志（commit log）

Kafka可以为一种外部的持久性日志的分布式系统提供服务。这种日志可以在节点间备份数据，并为故障节点数据回复提供一种重新同步的机制。Kafka中日志压缩功能为这种用法提供了条件。在这种用法中，Kafka类似于Apache BookKeeper项目。

#### Kafka的设计要点：

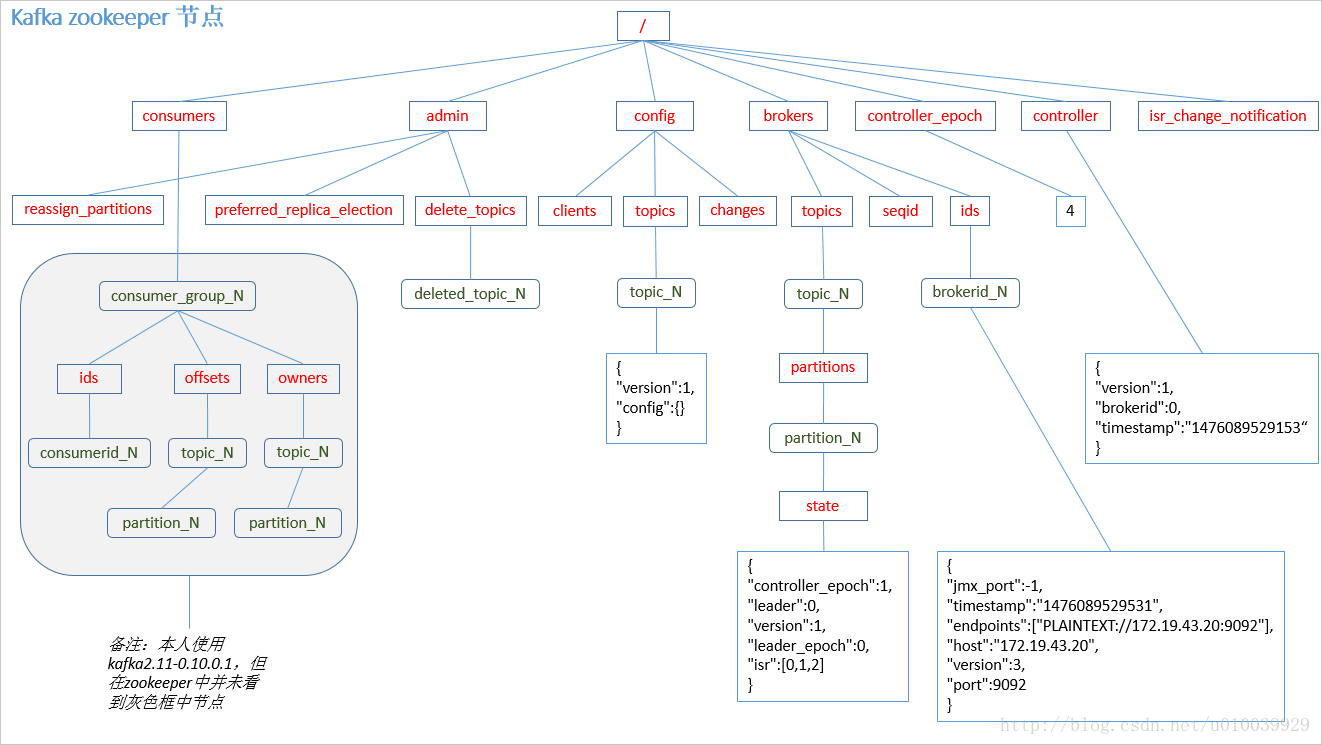
1、直接使用[**Linux**](http://lib.csdn.net/base/linux) 文件系统的cache，来高效缓存数据。

2、采用[**linux**](http://lib.csdn.net/base/linux) Zero-Copy提高发送性能。传统的数据发送需要发送4次上下文切换，采用sendfile系统调用之后，数据直接在内核态交换，系统上下文切换减少为2次。根据[**测试**](http://lib.csdn.net/base/softwaretest)结果，可以提高60%的数据发送性能。Zero-Copy详细的技术细节可以参考：<https://www.ibm.com/developerworks/linux/library/j-zerocopy/>

3、数据在磁盘上存取代价为O(1)。kafka以topic来进行消息管理，每个topic包含多个part（ition），每个part对应一个逻辑log，有多个segment组成。每个segment中存储多条消息，消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避免id到位置的额外映射。每个part在内存中对应一个index，记录每个segment中的第一条消息偏移。发布者发到某个topic的消息会被均匀的分布到多个part上（随机或根据用户指定的回调函数进行分布），broker收到发布消息往对应part的最后一个segment上添加该消息，当某个segment上的消息条数达到配置值或消息发布时间超过阈值时，segment上的消息会被flush到磁盘，只有flush到磁盘上的消息订阅者才能订阅到，segment达到一定的大小后将不会再往该segment写数据，broker会创建新的segment。

4、显式分布式，即所有的producer、broker和consumer都会有多个，均为分布式的。Producer和broker之间没有负载均衡机制。broker和consumer之间利用zookeeper进行负载均衡。所有broker和consumer都会在zookeeper中进行注册，且zookeeper会保存他们的一些元数据信息。如果某个broker和consumer发生了变化，所有其他的broker和consumer都会得到通知。

kafka zookeeper中的节点   
kafka 在 zookeeper 中的存储结构如下图所示：



#### **producer 发布消息**

**写入方式**

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。   
**消息路由**

 producer 发送消息到 broker 时，会根据分区[**算法**](http://lib.csdn.net/base/datastructure)选择将其存储到哪一个 partition。其路由机制为：

1. 指定了 patition，则直接使用；
2. 未指定 patition 但指定 key，通过对 key 的 value 进行hash 选出一个 patition
3. patition 和 key 都未指定，使用轮询选出一个 patition。

附上 [**Java**](http://lib.csdn.net/base/java) 客户端分区源码，一目了然：

//创建消息实例

public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) {

if (topic == null)

throw new IllegalArgumentException("Topic cannot be null");

if (timestamp != null && timestamp < 0)

throw new IllegalArgumentException("Invalid timestamp " + timestamp);

this.topic = topic;

this.partition = partition;

this.key = key;

this.value = value;

this.timestamp = timestamp;

}

//计算 patition，如果指定了 patition 则直接使用，否则使用 key 计算

private int partition(ProducerRecord<K, V> record, byte[] serializedKey , byte[] serializedValue, Cluster cluster) {

Integer partition = record.partition();

if (partition != null) {

List<PartitionInfo> partitions = cluster.partitionsForTopic(record.topic());

int lastPartition = partitions.size() - 1;

if (partition < 0 || partition > lastPartition) {

throw new IllegalArgumentException(String.format("Invalid partition given with record: %d is not in the range [0...%d].", partition, lastPartition));

}

return partition;

}

return this.partitioner.partition(record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster);

}

// 使用 key 选取 patition

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);

int numPartitions = partitions.size();

if (keyBytes == null) {

int nextValue = counter.getAndIncrement();

List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);

if (availablePartitions.size() > 0) {

int part = DefaultPartitioner.toPositive(nextValue) % availablePartitions.size();

return availablePartitions.get(part).partition();

} else {

return DefaultPartitioner.toPositive(nextValue) % numPartitions;

}

} else {

//对 keyBytes 进行 hash 选出一个 patition

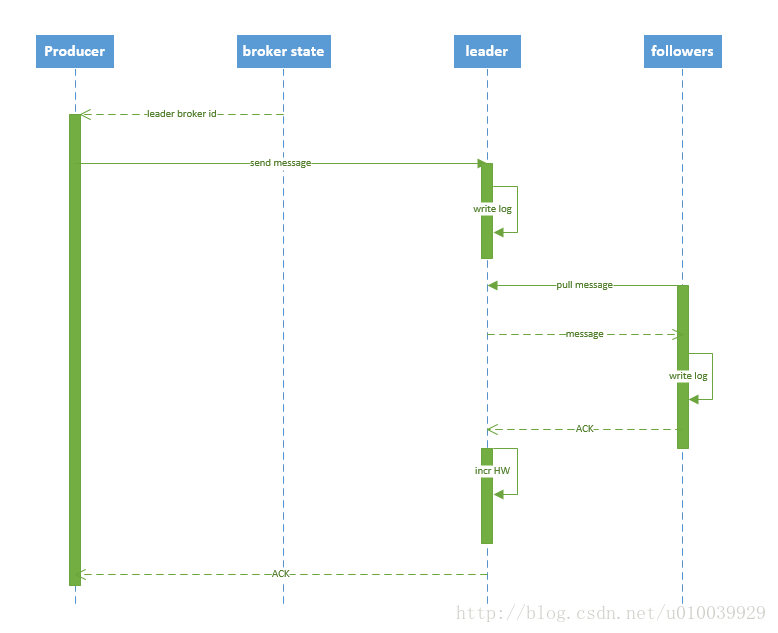
return DefaultPartitioner.toPositive(Utils.murmur2(keyBytes)) % numPartitions;

}

}

#### **写入流程**

producer 写入消息序列图如下所示：



1. producer 先从 zookeeper 的 “/brokers/…/state” 节点找到该 partition 的 leader
2. producer 将消息发送给该 leader
3. leader 将消息写入本地 log
4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK
5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW（high watermark，最后 commit 的 offset） 并向 producer 发送 ACK

#### **producer delivery guarantee**

一般情况下存在三种情况：

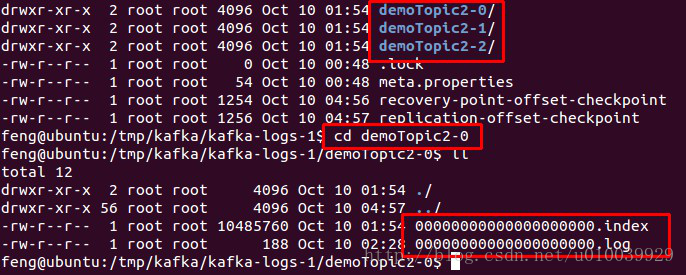
1. At most once 消息可能会丢，但绝不会重复传输
2. At least one 消息绝不会丢，但可能会重复传输
3. Exactly once 每条消息肯定会被传输一次且仅传输一次

        当 producer 向 broker 发送消息时，一旦这条消息被 commit，由于 replication 的存在，它就不会丢。但是如果 producer 发送数据给 broker 后，遇到网络问题而造成通信中断，那 Producer 就无法判断该条消息是否已经 commit。虽然 Kafka 无法确定网络故障期间发生了什么，但是 producer 可以生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了 Exactly once，但目前还并未实现。所以目前默认情况下一条消息从 producer 到 broker 是确保了 At least once，可通过设置 producer 异步发送实现At most once。

#### **broker 保存消息**

**存储方式**

物理上把 topic 分成一个或多个 patition（对应 server.properties 中的 num.partitions=3 配置），每个 patition 物理上对应一个文件夹（该文件夹存储该 patition 的所有消息和索引文件），如下：



#### **存储策略**

无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

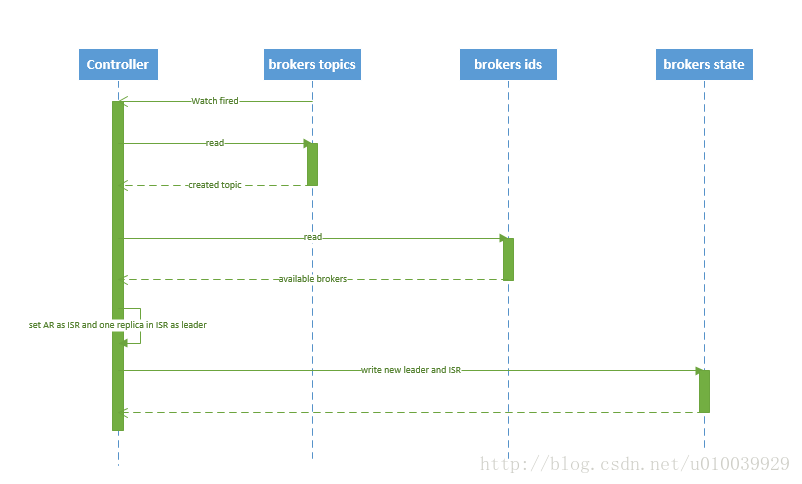
1. 基于时间：log.retention.hours=168
2. 基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。

#### **topic 创建与删除**

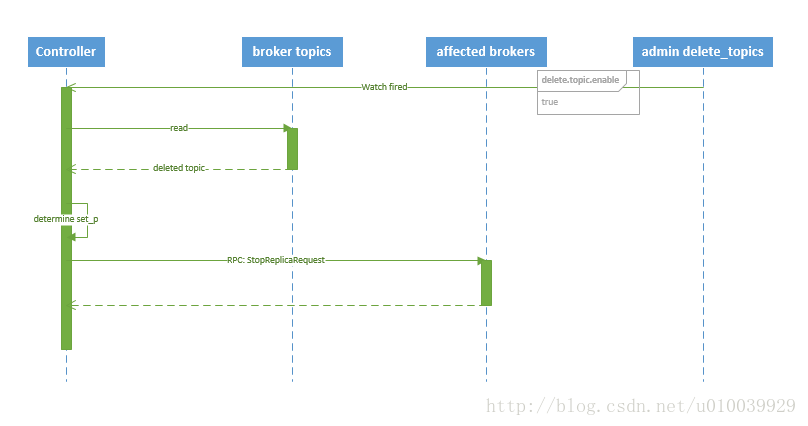
#### **创建 topic**

创建 topic 的序列图如下所示：



1. controller 在 ZooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被创建，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。
2. controller从 /brokers/ids 读取当前所有可用的 broker 列表，对于 set\_p 中的每一个 partition：   
   2.1 从分配给该 partition 的所有 replica（称为AR）中任选一个可用的 broker 作为新的 leader，并将AR设置为新的 ISR   
   2.2 将新的 leader 和 ISR 写入 /brokers/topics/[topic]/partitions/[partition]/state
3. controller 通过 RPC 向相关的 broker 发送 LeaderAndISRRequest。

#### **删除 topic**

删除 topic 的序列图如下所示：   
  
1. controller 在 zooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被删除，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。   
2. 若 delete.topic.enable=false，结束；否则 controller 注册在 /admin/delete\_topics 上的 watch 被 fire，controller 通过回调向对应的 broker 发送 StopReplicaRequest。

#### **kafka HA**

**replication**

同一个 partition 可能会有多个 replica（对应 server.properties 配置中的 default.replication.factor=N）。没有 replica 的情况下，一旦 broker 宕机，其上所有 patition 的数据都不可被消费，同时 producer 也不能再将数据存于其上的 patition。引入replication 之后，同一个 partition 可能会有多个 replica，而这时需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。

#### **Kafka 分配 Replica 的算法如下：**

1. 将所有 broker（假设共 n 个 broker）和待分配的 partition 排序
2. 将第 i 个 partition 分配到第（i mod n）个 broker 上
3. 将第 i 个 partition 的第 j 个 replica 分配到第（(i + j) mode n）个 broker上

**leader failover**

当 partition 对应的 leader 宕机时，需要从 follower 中选举出新 leader。在选举新leader时，一个基本的原则是，新的 leader 必须拥有旧 leader commit 过的所有消息。

kafka 在 zookeeper 中（/brokers/…/state）动态维护了一个 ISR（in-sync replicas），由3写入流程可知 ISR 里面的所有 replica 都跟上了 leader，只有 ISR 里面的成员才能选为 leader。对于 f+1 个 replica，一个 partition 可以在容忍 f 个 replica 失效的情况下保证消息不丢失。

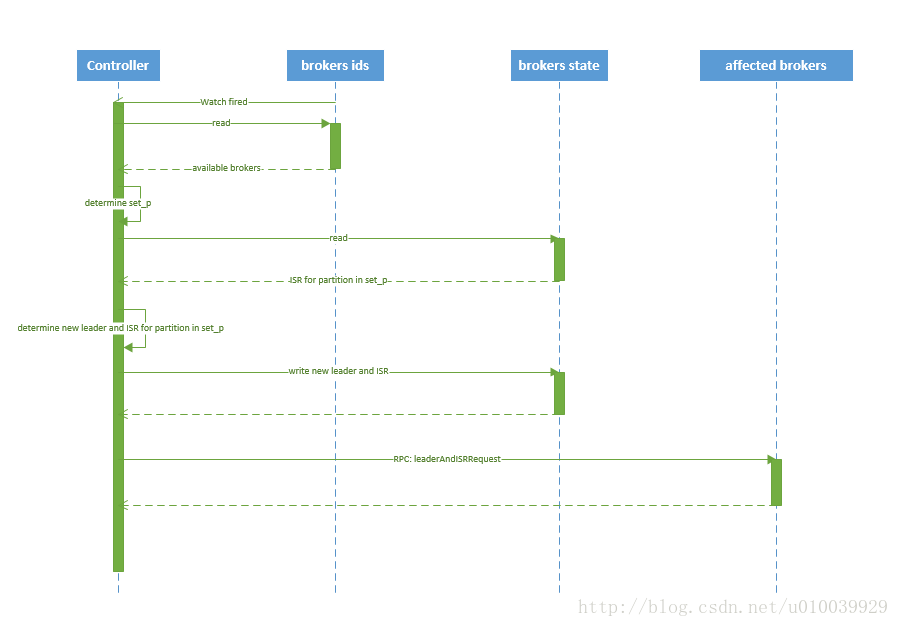
当所有 replica 都不工作时，有两种可行的方案：

1. 等待 ISR 中的任一个 replica 活过来，并选它作为 leader。可保障数据不丢失，但时间可能相对较长。
2. 选择第一个活过来的 replica（不一定是 ISR 成员）作为 leader。无法保障数据不丢失，但相对不可用时间较短。

kafka 0.8.\* 使用第二种方式。

kafka 通过 Controller 来选举 leader，流程请参考5.3节。

#### **broker failover**

kafka broker failover 序列图如下所示：   


1. controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker 宕机时 zookeeper 会 fire watch
2. controller 从 /brokers/ids 节点读取可用broker
3. controller决定set\_p，该集合包含宕机 broker 上的所有 partition
4. 对 set\_p 中的每一个 partition   
   4.1 从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR   
   4.2 决定新 leader（如4.3节所描述）   
   4.3 将新 leader、ISR、controller\_epoch 和 leader\_epoch 等信息写入 state 节点
5. 通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令

#### **controller failover**

当 controller 宕机时会触发 controller failover。每个 broker 都会在 zookeeper 的 “/controller” 节点注册 watcher，当 controller 宕机时 zookeeper 中的临时节点消失，所有存活的 broker 收到 fire 的通知，每个 broker 都尝试创建新的 controller path，只有一个竞选成功并当选为 controller。

当新的 controller 当选时，会触发 KafkaController.onControllerFailover 方法，在该方法中完成如下操作：

1. 读取并增加 Controller Epoch。
2. 在 reassignedPartitions Patch(/admin/reassign\_partitions) 上注册 watcher。
3. 在 preferredReplicaElection Path(/admin/preferred\_replica\_election) 上注册 watcher。
4. 通过 partitionStateMachine 在 broker Topics Patch(/brokers/topics) 上注册 watcher。
5. 若 delete.topic.enable=true（默认值是 false），则 partitionStateMachine 在 Delete Topic Patch(/admin/delete\_topics) 上注册 watcher。
6. 通过 replicaStateMachine在 Broker Ids Patch(/brokers/ids)上注册Watch。
7. 初始化 ControllerContext 对象，设置当前所有 topic，“活”着的 broker 列表，所有 partition 的 leader 及 ISR等。
8. 启动 replicaStateMachine 和 partitionStateMachine。
9. 将 brokerState 状态设置为 RunningAsController。
10. 将每个 partition 的 Leadership 信息发送给所有“活”着的 broker。
11. 若 auto.leader.rebalance.enable=true（默认值是true），则启动 partition-rebalance 线程。
12. 若 delete.topic.enable=true 且Delete Topic Patch(/admin/delete\_topics)中有值，则删除相应的Topic。

#### **consumer 消费消息**

consumer API

kafka 提供了两套 consumer API：

1. The high-level Consumer API
2. The SimpleConsumer API

其中 high-level consumer API 提供了一个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。

**The high-level consumer API**

high-level consumer API 提供了 consumer group 的语义，一个消息只能被 group 内的一个 consumer 所消费，且 consumer 消费消息时不关注 offset，最后一个 offset 由 zookeeper 保存。

使用 high-level consumer API 可以是多线程的应用，应当注意：

1. 如果消费线程大于 patition 数量，则有些线程将收不到消息
2. 如果 patition 数量大于线程数，则有些线程多收到多个 patition 的消息
3. 如果一个线程消费多个 patition，则无法保证你收到的消息的顺序，而一个 patition 内的消息是有序的

**The SimpleConsumer API**

如果你想要对 patition 有更多的控制权，那就应该使用 SimpleConsumer API，比如：

1. 多次读取一个消息
2. 只消费一个 patition 中的部分消息
3. 使用事务来保证一个消息仅被消费一次

但是使用此 API 时，partition、offset、broker、leader 等对你不再透明，需要自己去管理。你需要做大量的额外工作：

1. 必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息
2. 应用程序需要通过程序获知每个 Partition 的 leader 是谁
3. 需要处理 leader 的变更

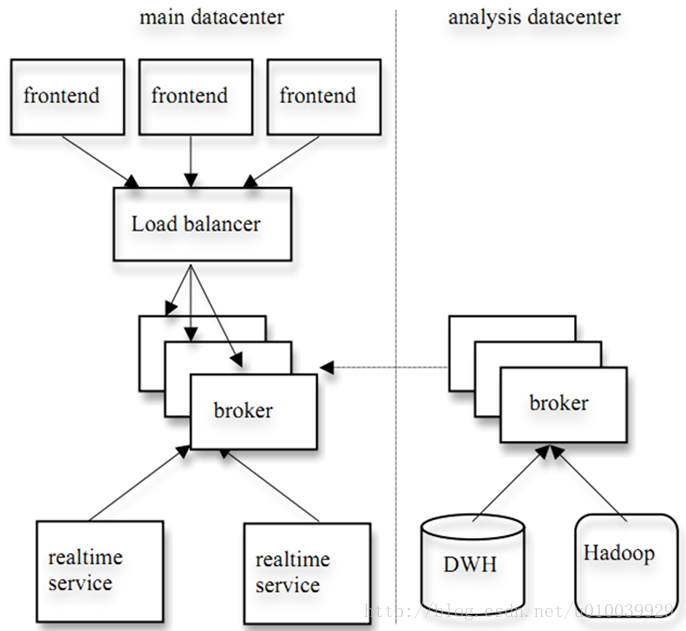
使用 SimpleConsumer API 的一般流程如下：

1. 查找到一个“活着”的 broker，并且找出每个 partition 的 leader
2. 找出每个 partition 的 follower
3. 定义好请求，该请求应该能描述应用程序需要哪些数据
4. fetch 数据
5. 识别 leader 的变化，并对之作出必要的响应

!   
以下针对 high-level Consumer API 进行说明。

consumer group

kafka 的分配单位是 patition。每个 consumer 都属于一个 group，一个 partition 只能被同一个 group 内的一个 consumer 所消费（也就保障了一个消息只能被 group 内的一个 consuemr 所消费），但是多个 group 可以同时消费这个 partition。

kafka 的设计目标之一就是同时实现离线处理和实时处理，根据这一特性，可以使用 [**Spark**](http://lib.csdn.net/base/spark)/Storm 这些实时处理系统对消息在线处理，同时使用 [**hadoop**](http://lib.csdn.net/base/hadoop) 批处理系统进行离线处理，还可以将数据备份到另一个数据中心，只需要保证这三者属于不同的 consumer group。如下图所示：   


消费方式

consumer 采用 pull 模式从 broker 中读取数据。

push 模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。它的目标是尽可能以最快速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 consumer 的消费能力以适当的速率消费消息。

对于 Kafka 而言，pull 模式更合适，它可简化 broker 的设计，consumer 可自主控制消费消息的速率，同时 consumer 可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

#### **consumer delivery guarantee**

如果将 consumer 设置为 autocommit，consumer 一旦读到数据立即自动 commit。如果只讨论这一读取消息的过程，那 Kafka 确保了 Exactly once。

但实际使用中应用程序并非在 consumer 读取完数据就结束了，而是要进行进一步处理，而数据处理与 commit 的顺序在很大程度上决定了consumer delivery guarantee：

1.读完消息先 commit 再处理消息。   
这种模式下，如果 consumer 在 commit 后还没来得及处理消息就 crash 了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对应于 At most once   
2.读完消息先处理再 commit。   
这种模式下，如果在处理完消息之后 commit 之前 consumer crash 了，下次重新开始工作时还会处理刚刚未 commit 的消息，实际上该消息已经被处理过了。这就对应于 At least once。   
3.如果一定要做到 Exactly once，就需要协调 offset 和实际操作的输出。   
精典的做法是引入两阶段提交。如果能让 offset 和操作输入存在同一个地方，会更简洁和通用。这种方式可能更好，因为许多输出系统可能不支持两阶段提交。比如，consumer 拿到数据后可能把数据放到 HDFS，如果把最新的 offset 和数据本身一起写到 HDFS，那就可以保证数据的输出和 offset 的更新要么都完成，要么都不完成，间接实现 Exactly once。（目前就 high-level API而言，offset 是存于Zookeeper 中的，无法存于HDFS，而SimpleConsuemr API的 offset 是由自己去维护的，可以将之存于 HDFS 中）

总之，Kafka 默认保证 At least once，并且允许通过设置 producer 异步提交来实现 At most once（见文章《kafka consumer防止数据丢失》）。而 Exactly once 要求与外部存储系统协作，幸运的是 kafka 提供的 offset 可以非常直接非常容易得使用这种方式。

#### **consumer rebalance**

当有 consumer 加入或退出、以及 partition 的改变（如 broker 加入或退出）时会触发 rebalance。consumer rebalance算法如下：   
1. 将目标 topic 下的所有 partirtion 排序，存于PT   
2. 对某 consumer group 下所有 consumer 排序，存于 CG，第 i 个consumer 记为 Ci   
3. N=size(PT)/size(CG)，向上取整   
4. 解除 Ci 对原来分配的 partition 的消费权（i从0开始）   
5. 将第i\*N到（i+1）\*N-1个 partition 分配给 Ci

在 0.8.\*版本，每个 consumer 都只负责调整自己所消费的 partition，为了保证整个consumer group 的一致性，当一个 consumer 触发了 rebalance 时，该 consumer group 内的其它所有其它 consumer 也应该同时触发 rebalance。这会导致以下几个问题：

1.Herd effect   
　　任何 broker 或者 consumer 的增减都会触发所有的 consumer 的 rebalance   
2.Split Brain   
　　每个 consumer 分别单独通过 zookeeper 判断哪些 broker 和 consumer 宕机了，那么不同 consumer 在同一时刻从 zookeeper 看到的 view 就可能不一样，这是由 zookeeper 的特性决定的，这就会造成不正确的 reblance 尝试。   
3. 调整结果不可控   
　　所有的 consumer 都并不知道其它 consumer 的 rebalance 是否成功，这可能会导致 kafka 工作在一个不正确的状态。

基于以上问题，kafka 设计者考虑在0.9.\*版本开始使用中心 coordinator 来控制 consumer rebalance，然后又从简便性和验证要求两方面考虑，计划在 consumer 客户端实现分配方案.

**注意事项**

producer 无法发送消息的问题

最开始在本机搭建了kafka伪集群，本地 producer 客户端成功发布消息至 broker。随后在服务器上搭建了 kafka 集群，在本机连接该集群，producer 却无法发布消息到 broker（奇怪也没有抛错）。最开始怀疑是 iptables 没开放，于是开放端口，结果还不行（又开始是代码问题、版本问题等等，倒腾了很久）。最后没办法，一项一项查看 server.properties 配置，发现以下两个配置：

# The address the socket server listens on. It will get the value returned from

# java.net.InetAddress.getCanonicalHostName() if not configured.

# FORMAT:

# listeners = security\_protocol://host\_name:port

# EXAMPLE:

# listeners = PLAINTEXT://your.host.name:9092

listeners=PLAINTEXT://:9092

　# Hostname and port the broker will advertise to producers and consumers. If not set,

　# it uses the value for "listeners" if configured. Otherwise, it will use the value

　# returned from java.net.InetAddress.getCanonicalHostName().

　#advertised.listeners=PLAINTEXT://your.host.name:9092

以上说的就是 advertised.listeners 是 broker 给 producer 和 consumer 连接使用的，如果没有设置，就使用 listeners，而如果 host\_name 没有设置的话，就使用 java[**.NET**](http://lib.csdn.net/base/dotnet).InetAddress.getCanonicalHostName() 方法返回的主机名。

修改方法：

1. listeners=PLAINTEXT://121.10.26.XXX:9092
2. advertised.listeners=PLAINTEXT://121.10.26.XXX:9092

# MySQL相关

#### 数据库优化：

（1）建库：建立索引；分区分表；使用固定长度的字段；限制字段长度

（2）I/O：增加缓冲区；级联查询，表文件的存储

（3）SQL：优化sql语句减少比较次数；限制返回的条目数-limit

（4）Java：反复使用的查询，preparedStatement预编译减少查询次数；二级缓存

#### msyql性能调优三件利器：

1.慢查询 （分析出现出问题的sql，show processlist 命令，显示哪些线程正在运行。）

2.Explain（查看执行计划，显示了mysql如何使用索引来处理select语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句。Mysql执行计划的的顺序：id相同，执行顺序由上至下；如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行。）

3.Profile（查询到 SQL 会执行多少时间, 并看出 CPU/Memory 使用量, 执行过程中 Systemlock, Table lock 花多少时间等等.）

#### MySQL存储引擎：MyISAM 和InnoDB

差别是Innodb 支持事务处理与外键和行级锁。

MyISAM不支持.MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快

innodb存储引擎通过预写事务日志的方式保障事物原子性一致性持久性，他包括redo日志和undo日志

redo日志在系统需要时对事物操作进行重做，系统宕机重启后，能够对内存中尚未持久化到磁盘的数据进行恢复

undo日志则能够在事务执行失败时，利用这些undo信息将数据还原到事务执行前的状态。

数据库索引有什么好处？创建索引可以大大提高系统的性能。

在经常需要搜索的列上，可以加快搜索的速度；

在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

在经常用在连接的列上，这 些列主要是一些外键，可以加快连接的速度；

在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

在经常需要排序的列上创 建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

#### Join查询：

Left Join（左联接）：以左表为中心，返回左表中符合条件的所有记录以及右表中联结字段相等的记录——当右表中无相应联接记录时，返回空值。

Right Join（右联接）：以右表为中心，返回右表中符合条件的所有记录以及左表中联结字段相等的记录——当左表中无相应联接记录时，返回空值。

Inner Join（等值连接） ：返回两个表中联结字段相等的行。

#### 乐观锁、悲观锁

    (1)悲观锁：

假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

MySQL InnoDB默认行级锁，通过开启排他锁(select…for update)的方式实现了悲观锁

//0.开始事务

begin;/begin work;/start transaction; (三者选一就可以)

//1.查询出商品信息

select status from t\_goods where id=1 for update;

//2.根据商品信息生成订单

insert into t\_orders (id,goods\_id) values (null,1);

//3.修改商品status为2

update t\_goods set status=2;

//4.提交事务

commit;/commit work;

    (2)乐观锁

假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。[1] 乐观锁不能解决脏读的问题。写比较多，经常产生冲突，上层应用会不断的进行retry。

写比较少，冲突很少发生，省去了锁的开销，加大了系统的整个吞吐量。

实现乐观锁2种方式：

1.使用数据版本（Version）记录机制实现

2.使用时间戳（timestamp）记录机制实现

select (id,status,version) from t\_goods where id=1;

>>id=1,status=1,version=1

update t\_goods set status=2,version=version+1 where id=1 and version=1;

>>YES:id=1,status=2,version=2

update t\_goods set status=2,version

#### 数据库中事务

事务是一系列操作组成的工作单元，该工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做。

MySQL默认隔离级别：可重复读-REPEATABLE-READ

四个隔离级别：1.    读未提交。2.    读已提交。3.    可重复读。4.    读序列化

#### ACID特性：

1.原子性（Atomicity）

即事务是不可分割的最小工作单元，事务内的操作要么全做，要么全不做

2.一致性（Consistency）

在事务执行前数据库的数据处于正确的状态，而事务执行完成后数据库的数据还是处于正确的状态，即数据完整性约束没有被破坏

如银行转帐，A转帐给B，必须保证A的钱一定转给B，一定不会出现A的钱转了但B没收到，否则数据库的数据就处于不一致（不正确）的状态

3.隔离性（Isolation）

并发事务执行之间无影响，在一个事务内部的操作对其他事务是不产生影响，这需要事务隔离级别来指定隔离性

4.持久性（Durability）

事务一旦执行成功，它对数据库的数据的改变必须是永久的，不会因比如遇到系统故障或断电造成数据不一致或丢失

锁是计算机协调多个进程或线程并发访问某一资源的机制，不同的数据库的锁机制大同小异。由于数据库资源是一种供许多用户共享的资源，所以如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。了解锁机制不仅可以使我们更有效的开发利用数据库资源，也使我们能够更好地维护数据库，从而提高数据库的性能。

上述三种锁的特性可大致归纳如下：

1 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，锁冲突的概率最高，并发度最低。

2 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，锁冲突的概率最低，并发度也最高。

3 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

     三种锁各有各的特点，若仅从锁的角度来说，表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如WEB应用；行级锁更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。

数据库行级锁：

优点：不同记录时减少冲突锁。事务回滚时减少改变数据。使长时间对单独的一行记录加锁成为可能。

缺点：比页级锁和表级锁消耗更多的内存。

----

#### JDBC连接数据库的程序，包含7个步骤：

 1、加载JDBC驱动程序：Class.forName("com.mysql.jdbc.Driver") ;

 2、提供JDBC连接的URL

jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=gbk ;

 3、创建数据库的连接

Connection con = DriverManager.getConnection(url , username , password ) ;

 4、创建一个Statement

Statement stmt = con.createStatement() ;

 5、执行SQL语句

Statement接口提供了三种执行SQL语句的方法：executeQuery、executeUpdate、execute

ResultSet rs = stmt.executeQuery("SELECT \* FROM ...") ;

int rows = stmt.executeUpdate("INSERT INTO ...") ;

boolean flag = stmt.execute(String sql) ;

 6、关闭JDBC对象

#### MyBatis

原理详解：

        MyBatis应用程序根据XML配置文件创建SqlSessionFactory，SqlSessionFactory在根据配置，配置来源于两个地方，一处是配置文件，一处是Java代码的注解，获取一个SqlSession。SqlSession包含了执行sql所需要的所有方法，可以通过SqlSession实例直接运行映射的sql语句，完成对数据的增删改查和事务提交等，用完之后关闭SqlSession。

#### 二级缓存

<cache  eviction="FIFO"  flushInterval="60000"  size="512"  readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。可用的收回策略有, 默认的是 LRU:

LRU – 最近最少使用的:移除最长时间不被使用的对象。

FIFO – 先进先出:按对象进入缓存的顺序来移除它们。

SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。

WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

#### MyBatis一级缓存，二级缓存：

一级缓存基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该Session中的所有 Cache 就将清空。

二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache、Hazelcast等。

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

MyBatis提供Cache 接口

public interface Cache {

  String getId();

  int getSize();

  void putObject(Object key, Object value);

  Object getObject(Object key);

  Object removeObject(Object key);

  void clear();

  ReadWriteLock getReadWriteLock();

}

Hibernate与Mybatis对比总结

    两者相同点

Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启执行事务和SQL语句。其中SessionFactoryBuider，SessionFactory，Session的生命周期都是差不多的。

Hibernate和MyBatis都支持JDBC和JTA事务处理。

    Mybatis优势

MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

MyBatis容易掌握，而Hibernate门槛较高。

    Hibernate优势

Hibernate的DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。

Hibernate对对象的维护和缓存要比MyBatis好，对增删改查的对象的维护要方便。

Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

    他人总结

Hibernate功能强大，数据库无关性好，O/R映射能力强，缺点就是学习门槛高，设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

# HBase

#### HBase协处理器

[**Hbase**](http://lib.csdn.net/base/hbase)作为列族[**数据库**](http://lib.csdn.net/base/mysql)最经常被人诟病的特性包括：1，无法轻易建立“二级索引”；2，难以执行求和、计数、排序等操作。比如，在旧版本的(<0.92)[**hbase**](http://lib.csdn.net/base/hbase)中，统计数据表的总行数，需要使用Counter方法，执行一次MapReduce Job才能得到。虽然HBase在数据存储层中集成了MapReduce，能够有效用于数据表的分布式计算。然而在很多情况下，做一些简单的相加或者聚合计算的时候，如果直接将计算过程放置在server端，能够减少通讯开销，从而获得很好的性能提升。于是，HBase在0.92之后引入了协处理器(coprocessors)，实现一些激动人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等。

HBase协处理器的灵感来自于Jeff Dean 09年的演讲，它根据该演讲实现了类似于bigtable的协处理器，包括以下特性:

每个表服务器的任意子表都可以运行代码

客户端的高层调用接口(客户端能够直接访问数据表的行地址，多行读写会自动分片成多个并行的RPC调用)

提供一个非常灵活的、可用于建立分布式服务的数据模型

能够自动化扩展、负载均衡、应用请求路由

HBase的协处理器灵感来自bigtable，但是实现细节不尽相同。HBase建立了一个框架，它为用户提供类库和运行时环境，使得他们的代码能够在HBase region server和master上处理。

##### **协处理器类型**

协处理器分两种类型，系统协处理器可以全局导入region server上的所有数据表，表协处理器即是用户可以指定一张表使用协处理器。协处理器框架为了更好支持其行为的灵活性，提供了两个不同方面的插件。一个是观察者（observer），类似于关系数据库的触发器。另一个是终端(endpoint)，动态的终端有点像存储过程。

##### **观察者(Observer)**

观察者的设计意图是允许用户通过插入代码来重载协处理器框架的upcall方法，而具体的事件触发的callback方法由HBase的核心代码来执行。协处理器框架处理所有的callback调用细节，协处理器自身只需要插入添加或者改变的功能。

以HBase0.92版本为例，它提供了三种观察者接口：

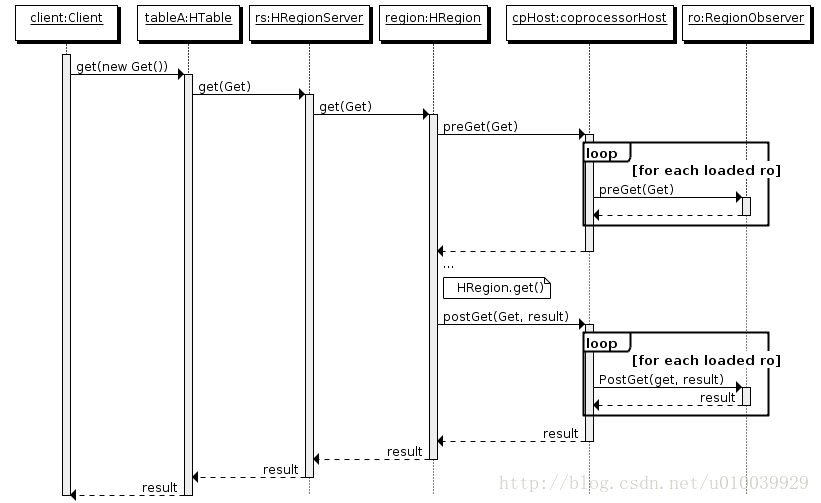
RegionObserver：提供客户端的数据操纵事件钩子：Get、Put、Delete、Scan等。

WALObserver：提供WAL相关操作钩子。

MasterObserver：提供DDL-类型的操作钩子。如创建、删除、修改数据表等。

这些接口可以同时使用在同一个地方，按照不同优先级顺序执行。用户可以任意基于协处理器实现复杂的HBase功能层。HBase有很多种事件可以触发观察者方法，这些事件与方法从HBase0.92版本起，都会集成在HBase API中。不过这些API可能会由于各种原因有所改动，不同版本的接口改动比较大，具体参考[**Java**](http://lib.csdn.net/base/java) Doc。

RegionObserver工作原理，如下图所示：



##### **终端(Endpoint)**

终端是动态RPC插件的接口，它的实现代码被安装在服务器端，从而能够通过HBase RPC唤醒。客户端类库提供了非常方便的方法来调用这些动态接口，它们可以在任意时候调用一个终端，它们的实现代码会被目标region远程执行，结果会返回到终端。用户可以结合使用这些强大的插件接口，为HBase添加全新的特性。终端的使用，如下面流程所示：

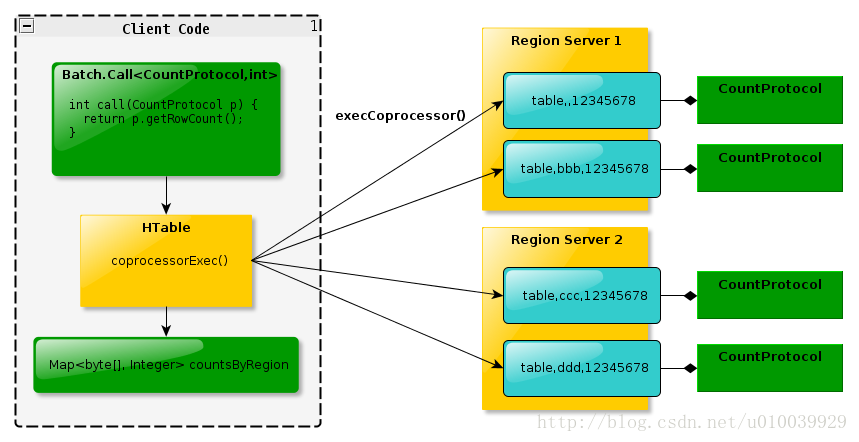
定义一个新的protocol接口，必须继承CoprocessorProtocol.

实现终端接口，该实现会被导入region环境执行。

继承抽象类BaseEndpointCoprocessor.

在客户端，终端可以被两个新的HBase Client API调用 。单个region：HTableInterface.coprocessorProxy(Class<T> protocol, byte[] row) 。rigons区域：HTableInterface.coprocessorExec(Class<T> protocol, byte[] startKey, byte[] endKey, Batch.Call<T,R> callable)

整体的终端调用过程范例，如下图所示：



**编程实践(Code Example)**

在该实例中，我们通过计算HBase表中行数的一个实例，来真实感受协处理器 的方便和强大。在旧版的HBase我们需要编写MapReduce代码来汇总数据表中的行数，在0.92以上的版本HBase中，只需要编写客户端的代码即可实现，非常适合用在WebService的封装上。

启用协处理器 Aggregation(Enable Coprocessor Aggregation)

有两个方法：1.启动全局aggregation，能过操纵所有的表上的数据。通过修改hbase-site.xml这个文件来实现，只需要添加如下代码：

<property>

<name>hbase.coprocessor.user.region.classes</name>

<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>

</property>

2.启用表aggregation，只对特定的表生效。通过HBase Shell 来实现。

(1) disable指定表

hbase> disable 'mytable'

(2) 添加aggregation

hbase>alter 'mytable', METHOD => 'table\_att','coprocessor'=>' |org.apache.Hadoop.hbase.coprocessor.AggregateImplementation||'

(3) 重启指定表

hbase> enable 'mytable'

首先从整体上了解一下开发 HBase 协处理器的流程。对于 Endpoint 类型的协处理器，其开发流程如下：

第一步是建立一个 [**Java**](http://lib.csdn.net/base/java) 工程；第二步是定义用户 Client HBase 通信的 RFC，采用 Protobuf 语言和工具完成定义；第三步是编写 HBase 协处理器的 Client 端和 Server 端代码，其中，Client 端代码负责调用协处理器并处理返回结果，Server 端代码将运行在 Region Server 上，实现具体的任务；最后需要对编译好的代码进行部署和[**测试**](http://lib.csdn.net/base/softwaretest)。

对于 Observer 类型的协处理器，不需要定义 RPC，也不需要开发客户端代码。当相应的事件发生时，Observer 代码将自动在 Server 端执行。因此仅仅需要编写 Server 端的代码即可。

public class MyAggregationClient {

private static final byte[] TABLE\_NAME = Bytes.toBytes("mytable");

private static final byte[] CF = Bytes.toBytes("vent");

public static void main(String[] args) throws Throwable {

Configuration customConf = new Configuration();

customConf.setStrings("hbase.zookeeper.quorum",

"node0,node1,node2");

//提高RPC通信时长

customConf.setLong("hbase.rpc.timeout", 600000);

//设置Scan缓存

customConf.setLong("hbase.client.scanner.caching", 1000);

Configuration configuration = HBaseConfiguration.create(customConf);

AggregationClient aggregationClient = new AggregationClient(

configuration);

Scan scan = new Scan();

//指定扫描列族，唯一值

scan.addFamily(CF);

long rowCount = aggregationClient.rowCount(TABLE\_NAME, null, scan);

System.out.println("row count is " + rowCount);

}

}

**典型例子**

协处理器其中的一个作用是使用Observer创建二级索引。先举个实际例子：   
我们要查询指定店铺指定客户购买的订单，首先有一张订单详情表，它以被处理后的订单id作为rowkey；其次有一张以客户nick为rowkey的索引表，结构如下：

rowkey family   
dp\_id+buy\_nick1 tid1:null tid2:null …   
dp\_id+buy\_nick2 tid3:null   
…   
该表可以通过Coprocessor来构建，实例代码：

public class TestCoprocessor extends BaseRegionObserver {

@Override

public void prePut(final ObserverContext<RegionCoprocessorEnvironment> e,

final Put put, final WALEdit edit, final boolean writeToWAL)

throws IOException {

Configuration conf = new Configuration();

HTable table = new HTable(conf, "index\_table");

List<KeyValue> kv = put.get("data".getBytes(), "name".getBytes());

Iterator<KeyValue> kvItor = kv.iterator();

while (kvItor.hasNext()) {

KeyValue tmp = kvItor.next();

Put indexPut = new Put(tmp.getValue());

indexPut.add("index".getBytes(), tmp.getRow(), Bytes.toBytes(System.currentTimeMillis()));

table.put(indexPut);

}

table.close();

}

}

即继承BaseRegionObserver类，实现prePut方法，在插入订单详情表之前，向索引表插入索引数据。

**索引表的使用**

先在索引表get索引表，获取tids，然后根据tids查询订单详情表。   
当有多个查询条件（多张索引表），根据逻辑运算符（and 、or）确定tids。

**使用时注意**

1.索引表是一张普通的hbase表，为安全考虑需要开启Hlog记录日志。   
2.索引表的rowkey最好是不可变量，避免索引表中产生大量的脏数据。   
3.如上例子，column是横向扩展的（宽表），rowkey设计除了要考虑region均衡，也要考虑column数量，即表不要太宽。建议不超过3位数。   
4.如上代码，一个put操作其实是先后向两张表put数据，为保证一致性，需要考虑异常处理，建议异常时重试。

**效率情况**

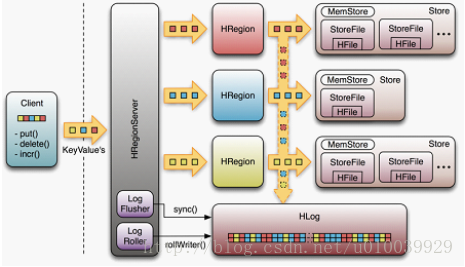
put操作效率不高，如上代码，每插入一条数据需要创建一个新的索引表连接（可以使用htablepool优化），向索引表插入数据。即耗时是双倍的，对hbase的集群的压力也是双倍的。当索引表有多个时，压力会更大。   
查询效率比filter高，毫秒级别，因为都是rowkey的查询。   
如上是估计的效率情况，需要根据实际业务场景和集群情况而定，最好做预先测试。

**Coprocessor二级索引方案优劣**

优点：在put压力不大、索引region均衡的情况下，查询很快。   
缺点：业务性比较强，若有多个字段的查询，需要建立多张索引表，需要保证多张表的数据一致性，且在hbase的存储和内存上都会有更高的要求。

#### HBase WAL

WAL(Write-Ahead-Log)是HBase的RegionServer在处理数据插入和删除的过程中用来记录操作内容的一种日志。大致过程如下图所示，首先客户端启动一个操作来修改数据，每一个修改都封装到KeyValue对象实例中，并通过RPC调用发送到含有匹配Region的HRegionServer。一旦KeyValue到达，它们就会被发送管理相应行的HRegion实例。数据被写到WAL，然后被放入到实际拥有记录的存储文件的MemStore中。同时还会检查MemStore是否满了，如果满了就会被刷写到磁盘中去。



默认是每个RegionServer有1个WAL，在HBase1.0开始支持多个WALHBASE-5699,这样可以提高写入的吞吐量。配置参数为hbase.wal.provider=multiwal，支持的值还有defaultProvider和filesystem(这2个是同样的实现)。   
WAL的持久化的级别有如下几种：

SKIP\_WAL：不写wal日志,这种可以较大提高写入的性能，但是会存在数据丢失的危险，只有在大批量写入的时候才使用(出错了可以重新运行)，其他情况不建议使用。

ASYNC\_WAL：异步写入

SYNC\_WAL：同步写入wal日志文件，保证数据写入了DataNode节点。

FSYNC\_WAL: 目前不支持了，表现是与SYNC\_WAL是一致的

USE\_DEFAULT: 如果没有指定持久化级别，则默认为USE\_DEFAULT, 这个为使用HBase全局默认级别(SYNC\_WAL)

##### WAL写入：

与WAL相关的几个主要类：   
1. WALKey: wal日志的key，包括 regionName： 日志所属的region tablename：日志所属的表，writeTime：日志写入时间，clusterIds:cluster的id，在数据复制的时候会用到。   
2. WALEdit: 在hbase的事务日志中记录一系列的修改的一条事务日志。另外WALEdit实现了Writable接口，可用于序列化处理。   
3. FSHLog: WAL的实现类，负责将数据写入文件系统

对每个wal的写入使用的是**多生产者单消费者的模式**，这里使用到了disruptor框架，将**WALKey**和**WALEdit**信息封装为**FSWALEntry**，然后通过RingBufferTruck放入RingBuffer中。

接下来看hlog的写入流程，分为以下3步：

日志写入缓存：由rpcHandler将日志信息写入缓存ringBuffer.

缓存数据写入文件系统：每个FSHLog有一个线程负责将数据写入文件系统(HDFS)

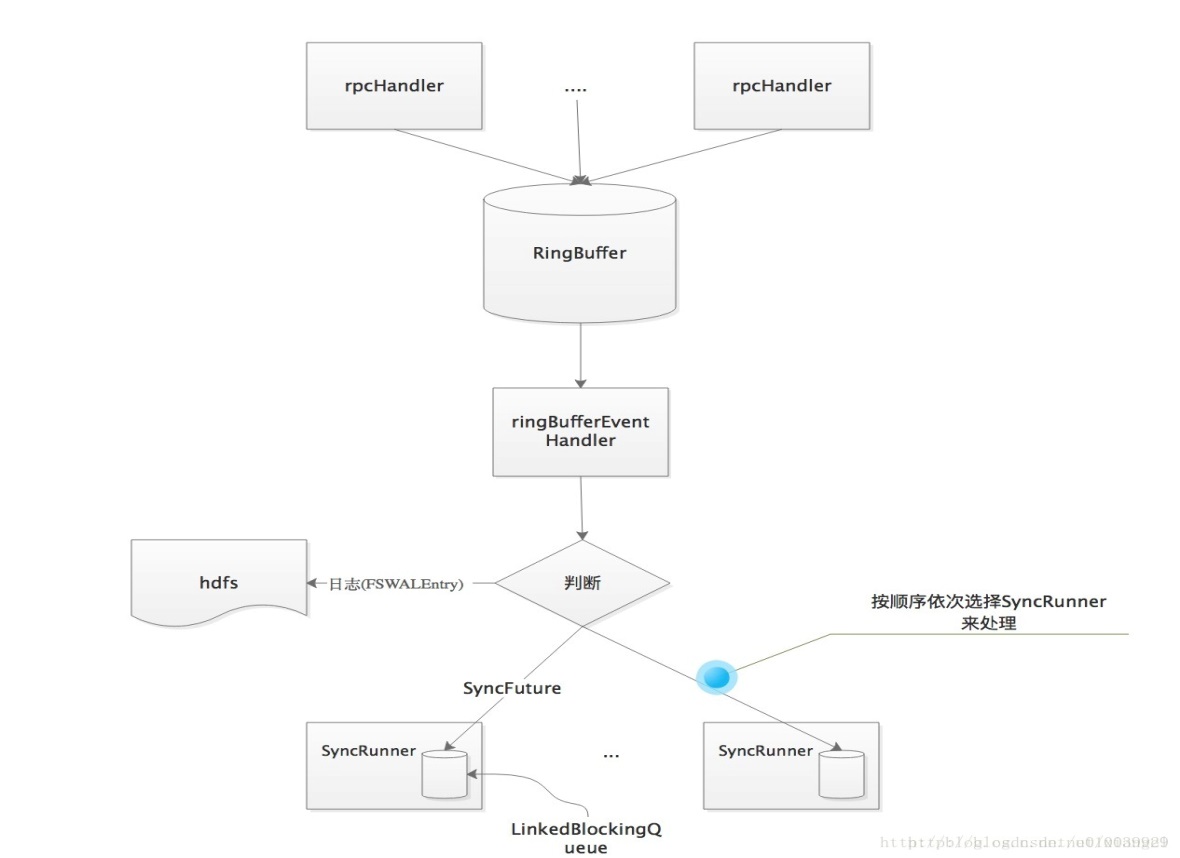
数据同步：如果操作的持久化级别为(SYNC\_WAL或者USE\_DEFAULT 则需进行数据同步处理

下面来详细说明一下各类线程是如何配合来实现这几步操作的，

rpcHandler线程负责将日志信息(FSWALEntry)写入缓存RingBbuffer，在操作日志写完后，rpcHandler会调用wal的sync方法，进行数据同步，其实际处理为写入一个SyncFuture到RingBuffer，然后blocking一直到syncFuture处理完成。

wal线程从缓存RingBuffer中取数据，如果为日志(FSWALEntry)就调用Writer将数据写入文件系统，如果为SyncFuture,则由专门的同步线程来进行同步处理。

整体处理流程图如下：



##### HLog的写入

wal写入文件系统是通过Writer来写入的，其实际类为ProtobufLogWriter，使用的是Protobuf的格式持久化处理。使用Protobuf格式有如下优势：

性能较高

结构更加紧凑，节省空间

方便扩展以及支持其他语言，通过其他语言来解析日志。

写入的日志中是按WALKey和WALEdit来依次存储的（具体内容见前面WALKey和WALEdit类的说明），另外还将WALKey和WALEdit分别进行了压缩处理。

WAL同步过程

每个wal中有一个RingBufferEventHandler对象，其中用数组管理着多个SyncRunner线程(由参数hbase.regionserver.hlog.syncer.count配置，默认5)来进行同步处理，每个SyncRunner对象里面有一个LinkedBlockingQueue(syncFutures,大小为参数{hbase.regionserver.handler.count默认值200}\*3   
另外这里的SyncFuture是每个rpcHandler线程拥有一个，由wal中的private final Map

class RingBufferEventHandler implements EventHandler<RingBufferTruck>, LifecycleAware {

private final SyncRunner [] syncRunners;

private final SyncFuture [] syncFutures;

private volatile int syncFuturesCount = 0;

private volatile SafePointZigZagLatch zigzagLatch;

private Exception exception = null;

private int syncRunnerIndex;

RingBufferEventHandler(final int syncRunnerCount, final int maxHandlersCount) {

this.syncFutures = new SyncFuture[maxHandlersCount];

this.syncRunners = new SyncRunner[syncRunnerCount];

for (int i = 0; i < syncRunnerCount; i++) {

this.syncRunners[i] = new SyncRunner("sync." + i, maxHandlersCount);

}

}

......

}

private class SyncRunner extends HasThread {

private volatile long sequence;

private final BlockingQueue<SyncFuture> syncFutures;

private volatile SyncFuture takeSyncFuture = null;

SyncRunner(final String name, final int maxHandlersCount) {

super(name);

this.syncFutures = new LinkedBlockingQueue<SyncFuture>(maxHandlersCount \* 3);

}

......

}

在处理ringBuffer中的syncFuture时，不是每有一个就提交到syncRunner处理，而是按批来处理的，这里的批分2种情况：

从ringBuffer中取到的一批数据(为提高效率，在disruptor框架中是按批从ringBuffer中取数据的，具体的请看disruptor的相关文档)，如果这批数据中的syncFuture个数<{hbase.regionserver.handler.count默认值200}，则按一批处理，如果这一批数据中的syncFuture个数>={hbase.regionserver.handler.count默认值200}个数，则按{hbase.regionserver.handler.count默认值200}分批处理。

如果达到了批大小，就从syncRunner数组中顺序选择下一个SyncRunner,将这批数据插入该SyncRunner的BlockingQueue中。最后由SyncRunner线程进行hdfs文件同步处理。为保证数据的不丢失，rpc请求需要保证wal日志写入成功后才能返回，这里HBase做了一系列的优化处理的操作。

##### WAL滚动

通过wal日志切换，可以避免产生单独的过大的wal日志文件，便于后续的日志清理(可以将过期日志文件直接删除)另外如果需要使用日志进行恢复时，也可以同时解析多个小的日志文件，缩短恢复所需时间。   
wal触发切换的场景有如下几种：

SyncRunner线程在处理日志同步后，如果有异常发生，就会调用requestLogRoll发起日志滚动请求

SyncRunner线程在处理日志同步后, 检查当前在写的wal的日志大小是否超过配置{hbase.regionserver.hlog.blocksize默认为hdfs目录块大小}\*{hbase.regionserver.logroll.multiplier默认0.95}，超过后同样调用requestLogRoll发起日志滚动请求

每个RegionServer有一个LogRoller线程会定期滚动日志，滚动周期由参数{hbase.regionserver.logroll.period默认值1个小时}控制

这里前面2种场景调用requestLogRoll发起日志滚动请求，最终也是通过LogRoller来执行日志滚动的操作。

##### WAL失效

当memstore中的数据刷新到hdfs后，那对应的wal日志就不需要了，FSHLog中有记录当前memstore中各region对应的最老的sequenceId，如果一个日志中的各个region的操作的最新的sequenceId均小于wal中记录的各个需刷新的region的最老sequenceId，说明该日志文件就不需要了，于是就会将该日志文件从./WALs目录移动到./oldWALs目录。这块是在前面日志滚动完成后调用cleanOldLogs来处理的。

WAL删除

由于wal日志还会用于跨集群的同步处理，所以wal日志失效后并不会立即删除，而是移动到oldWALs目录。由HMaster中的LogCleaner这个Chore线程来负责wal日志的删除，在LogCleaner内部通过参数{hbase.master.logcleaner.plugins}以插件的方式来筛选出可以删除的日志文件。目前配置的插件有ReplicationLogCleaner、SnapshotLogCleaner和TimeToLiveLogCleaner

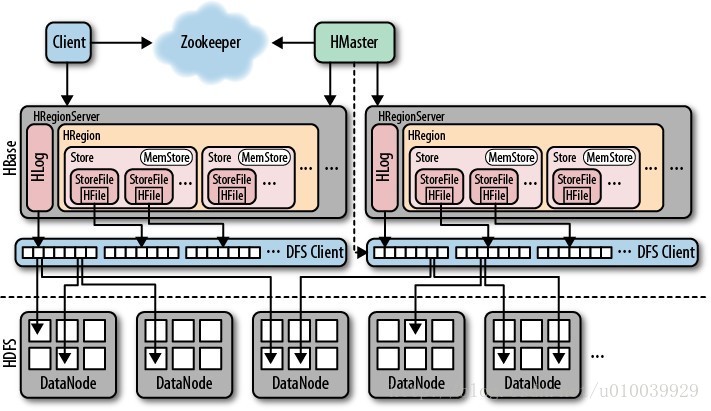
TimeToLiveLogCleaner: 日志文件最后修改时间在配置参数{hbase.master.logcleaner.ttl默认600秒}之前的可以删除

ReplicationLogCleaner:如果有跨集群数据同步的需求，通过该Cleaner来保证那些在同步中的日志不被删除

SnapshotLogCleaner: 被表的snapshot使用到了的wal不被删除

#### HBase Split

HBase Split是hbase根据一定的触发条件和一定的分裂策略将HBase的一个region进行分裂成两个子region并对父region进行清除处理的过程。Region是HBase中一个非常核心的组织单元，所有的region组成了整个HBase集群，如下面的HBase的体系结构所示：



在HBase中，split其实是进行sharding的一种技术手段，通过HBase的split条件和split策略，将region进行合理的split，再通过HBase的balance策略，将分裂的region负载均衡到各个regionserver上，最大化的发挥分布式系统的优点。HBase这种自动的sharding技术比传统的数据库sharding要省事的多，减轻了维护的成本，但是这样也会给HBase带来额外的IO开销，因此在很多系统中如果能很好的预计rowkey的分布和数据增长情况，可以通过**预先分区**，事先将region分配好，再将HBase的自动分区禁掉。

##### Region split的大概流程如下：

1.region先更改ZK中该region的状态为SPLITING。   
2.Master检测到region状态改变。   
3.region会在存储目录下新建.split文件夹用于保存split后的daughter region信息。   
4.Parent region关闭数据写入并触发flush操作，保证所有写入Parent region的数据都能持久化。   
5.在.split文件夹下新建两个region，称之为daughter A、daughter B。   
6.Daughter A、Daughter B拷贝到HBase根目录下，形成两个新的region。   
7.Parent region通知修改.META.表后下线，不再提供服务。   
8.Daughter A、Daughter B上线，开始向外提供服务。   
9.如果开启了balance\_switch服务，split后的region将会被重新分布。

上面1 ~ 9就是region split的整个过程，split过程非常快，速度基本会在秒级内，那么在这么快的时间内，region中的数据怎么被重新组织的？其实，split只是简单的把region从逻辑上划分成两个，并没有涉及到底层数据的重组，split完成后，Parent region并没有被销毁，只是被做下线处理，不再对外部提供服务。而新产生的region Daughter A 和 Daughter B，内部的数据只是简单的到Parent region数据的索引，Parent region数据的清理在Daughter A和Daughter B进行major compact以后，发现已经没有到其内部数据的索引后，Parent region才会被真正的清理。

##### **HBase Split触发条件**

1 Pre-splitting   
当一个table刚被创建的时候，Hbase默认的分配一个region给table。也就是说这个时候，所有的读写请求都会访问到同一个regionServer的同一个region中，这个时候就达不到负载均衡的效果了，集群中的其他regionServer就可能会处于比较空闲的状态。解决这个问题可以用pre-splitting，在创建table的时候就配置好，生成多个region。

在table初始化的时候如果不配置的话，Hbase是不知道如何去split region的，因为Hbase不知道应该那个row key可以作为split的开始点。如果我们可以大概预测到row key的分布，我们可以使用pre-spliting来帮助我们提前split region。不过如果我们预测得不准确的话，还是可能导致某个region过热，被集中访问，不过还好我们还有auto-split。最好的办法就是首先预测split的切分点，做pre-splitting,然后后面让auto-split来处理后面的负载均衡。

Hbase自带了两种pre-split的算法，分别是 HexStringSplit 和 UniformSplit 。如果我们的row key是十六进制的字符串作为前缀的，就比较适合用HexStringSplit，作为pre-split的算法。例如，我们使用HexHash(prefix)作为row key的前缀，其中Hexhash为最终得到十六进制字符串的hash算法。我们也可以用我们自己的split算法。

2 Auto splitting

1.当memstore flush操作后，HRegion写入新的HFile，有可能产生较大的HFile，HBase就会调用CompactSplitThread.requestSplit判断是否需要split操作。

2.HStore刚刚进行完compact操作后有可能产生较大的HFile，当满足HBase的某一分裂策略后就会进行split操作。

HBase默认有三种自动split的策略，ConstantSizeRegionSplitPolicy,IncreasingToUpperBoundRegionSplitPolicy还有 KeyPrefixRegionSplitPolicy。在0.94版本之前ConstantSizeRegionSplitPolicy 是默认和唯一的split策略。当某个store（对应一个column family）的大小大于配置值 ‘hbase.hregion.max.filesize’的时候（默认10G）region就会自动分裂。而0.94版本中，IncreasingToUpperBoundRegionSplitPolicy 是默认的split策略。这个策略中，最小的分裂大小和table的某个region server的region 个数有关，当store file的大小大于如下公式得出的值的时候就会split，公式如下

Min (R^2 \* “hbase.hregion.memstore.flush.size”, “hbase.hregion.max.filesize”) R为同一个table中在同一个region server中region的个数。

例如:

hbase.hregion.memstore.flush.size 默认值 128MB。

hbase.hregion.max.filesize默认值为10GB 。

如果初始时R=1,那么Min(128MB,10GB)=128MB,也就是说在第一个flush的时候就会触发分裂操作。

当R=2的时候Min(2\*2\*128MB,10GB)=512MB ,当某个store file大小达到512MB的时候，就会触发分裂。

如此类推，当R=9的时候，store file 达到10GB的时候就会分裂，也就是说当R>=9的时候，store file 达到10GB的时候就会分裂。

split 点都位于region中row key的中间点。KeyPrefixRegionSplitPolicy可以保证相同的前缀的row保存在同一个region中。指定rowkey前缀位数划分region，通过读取 KeyPrefixRegionSplitPolicy.prefix\_length 属性，该属性为数字类型，表示前缀长度，在进行split时，按此长度对splitPoint进行截取。此种策略比较适合固定前缀的rowkey。当table中没有设置该属性，指定此策略效果等同与使用IncreasingToUpperBoundRegionSplitPolicy。我们可以通过配置 hbase.regionserver.region.split.policy 来指定split策略，我们也可以写我们自己的split策略。

3 Forced Splits   
当HBaseAdmin手动发起split时，也会触发split操作。 、

##### **split策略**

1. 采用ConstantSizeRegionSplitPolicy策略，即storefile固定大小策略：

在0.94版本之前ConstantSizeRegionSplitPolicy 是默认和唯一的split策略。当某个storefile（对应一个columnfamily）的大小大于配置值 ‘hbase.hregion.max.filesize’的时候（默认DEFAULT\_MAX\_FILE\_SIZE = 10 \* 1024 \* 1024 \*1024L=10G）region就会自动分裂:

对应的源代码如下：

@Override

protected void configureForRegion(HRegion region) {

super.configureForRegion(region);

Configuration conf = getConf();

HTableDescriptor desc = region.getTableDesc();

if (desc != null) {

this.desiredMaxFileSize = desc.getMaxFileSize();

}

if (this.desiredMaxFileSize <= 0) {

this.desiredMaxFileSize = conf.getLong(HConstants.HREGION\_MAX\_FILESIZE,

HConstants.DEFAULT\_MAX\_FILE\_SIZE);

}

double jitter = conf.getDouble("hbase.hregion.max.filesize.jitter", 0.25D);

this.jitterRate = (RANDOM.nextFloat() - 0.5D) \* jitter;

long jitterValue = (long) (this.desiredMaxFileSize \* this.jitterRate);

// make sure the long value won't overflow with jitter

if (this.jitterRate > 0 && jitterValue > (Long.MAX\_VALUE - this.desiredMaxFileSize)) {

this.desiredMaxFileSize = Long.MAX\_VALUE;

} else {

this.desiredMaxFileSize += jitterValue;

}

}

@Override

protected boolean shouldSplit() {

boolean force = region.shouldForceSplit();

boolean foundABigStore = false;

for (Store store : region.getStores()) {

// If any of the stores are unable to split (eg they contain reference files)

// then don't split

if ((!store.canSplit())) {

return false;

}

// Mark if any store is big enough

if (store.getSize() > desiredMaxFileSize) {

foundABigStore = true;

}

}

return foundABigStore || force;

}

2采用IncreasingToUpperBoundRegionSplitPolicy策略，即根据region数来决定：

0.94.0（包含）之后，默认采用此策略，即当同一table在同一regionserver上的region数量在[0,100)之间时按照如下的计算公式算，否则按照上一策略计算：

Min (R^2\* “hbase.hregion.memstore.flush.size”\*2, “hbase.hregion.max.filesize”)R为同一个table中在同一个regionserver中region的个数，hbase.hregion.memstore.flush.size默认为128M，hbase.hregion.max.filesize默认为10G。

@Override

protected void configureForRegion(HRegion region) {

super.configureForRegion(region);

Configuration conf = getConf();

initialSize = conf.getLong("hbase.increasing.policy.initial.size", -1);

if (initialSize > 0) {

return;

}

HTableDescriptor desc = region.getTableDesc();

if (desc != null) {

initialSize = 2 \* desc.getMemStoreFlushSize();

}

if (initialSize <= 0) {

initialSize = 2 \* conf.getLong(HConstants.HREGION\_MEMSTORE\_FLUSH\_SIZE,

HTableDescriptor.DEFAULT\_MEMSTORE\_FLUSH\_SIZE);

}

}

@Override

protected boolean shouldSplit() {

boolean force = region.shouldForceSplit();

boolean foundABigStore = false;

// Get count of regions that have the same common table as this.region

int tableRegionsCount = getCountOfCommonTableRegions();

// Get size to check

long sizeToCheck = getSizeToCheck(tableRegionsCount);

for (Store store : region.getStores()) {

// If any of the stores is unable to split (eg they contain reference files)

// then don't split

if (!store.canSplit()) {

return false;

}

// Mark if any store is big enough

long size = store.getSize();

if (size > sizeToCheck) {

LOG.debug("ShouldSplit because " + store.getColumnFamilyName() + " size=" + size

+ ", sizeToCheck=" + sizeToCheck + ", regionsWithCommonTable="

+ tableRegionsCount);

foundABigStore = true;

}

}

return foundABigStore | force;

}

/\*\*

\* @return Count of regions on this server that share the table this.region

\* belongs to

\*/

private int getCountOfCommonTableRegions() {

RegionServerServices rss = region.getRegionServerServices();

// Can be null in tests

if (rss == null) {

return 0;

}

TableName tablename = region.getTableDesc().getTableName();

int tableRegionsCount = 0;

try {

List<Region> hri = rss.getOnlineRegions(tablename);

tableRegionsCount = hri == null || hri.isEmpty() ? 0 : hri.size();

} catch (IOException e) {

LOG.debug("Failed getOnlineRegions " + tablename, e);

}

return tableRegionsCount;

}

两种用户自定义策略：KeyPrefixRegionSplitPolicy和DelimitedKeyPrefixRegionSplitPolicy，这两种策略是IncreasingToUpperBoundRegionSplitPolicy策略的具体实现。

**KeyPrefixRegionSplitPolicy**

KeyPrefixRegionSplitPolicy策略，即根据rowkey指定长度的前缀来划分region：

即保证相同的前缀的row保存在同一个region中。指定rowkey前缀位数划分region，通过读取 KeyPrefixRegionSplitPolicy.prefix\_length 属性，该属性为数字类型，表示前缀长度，在进行split时，按此长度对splitPoint进行截取。此种策略比较适合固定前缀的rowkey。当table中没有设置该属性，指定此策略效果等同与使用IncreasingToUpperBoundRegionSplitPolicy。

@Override

protected void configureForRegion(HRegion region) {

super.configureForRegion(region);

prefixLength = 0;

// read the prefix length from the table descriptor

String prefixLengthString = region.getTableDesc().getValue(

PREFIX\_LENGTH\_KEY);

if (prefixLengthString == null) {

//read the deprecated value

prefixLengthString = region.getTableDesc().getValue(PREFIX\_LENGTH\_KEY\_DEPRECATED);

if (prefixLengthString == null) {

LOG.error(PREFIX\_LENGTH\_KEY + " not specified for table "

+ region.getTableDesc().getTableName()

+ ". Using default RegionSplitPolicy");

return;

}

}

try {

prefixLength = Integer.parseInt(prefixLengthString);

} catch (NumberFormatException nfe) {

/\* Differentiate NumberFormatException from an invalid value range reported below. \*/

LOG.error("Number format exception when parsing " + PREFIX\_LENGTH\_KEY + " for table "

+ region.getTableDesc().getTableName() + ":"

+ prefixLengthString + ". " + nfe);

return;

}

if (prefixLength <= 0) {

LOG.error("Invalid value for " + PREFIX\_LENGTH\_KEY + " for table "

+ region.getTableDesc().getTableName() + ":"

+ prefixLengthString + ". Using default RegionSplitPolicy");

}

}

**DelimitedKeyPrefixRegionSplitPolicy策略**

保证以分隔符前面的前缀为splitPoint，保证相同RowKey前缀的数据在一个Region中。

@Override

protected void configureForRegion(HRegion region) {

super.configureForRegion(region);

// read the prefix length from the table descriptor

String delimiterString = region.getTableDesc().getValue(DELIMITER\_KEY);

if (delimiterString == null || delimiterString.length() == 0) {

LOG.error(DELIMITER\_KEY + " not specified for table " + region.getTableDesc().getTableName() +

". Using default RegionSplitPolicy");

return;

}

delimiter = Bytes.toBytes(delimiterString);

}

@Override

protected byte[] getSplitPoint() {

byte[] splitPoint = super.getSplitPoint();

if (splitPoint != null && delimiter != null) {

//find the first occurrence of delimiter in split point

int index = com.google.common.primitives.Bytes.indexOf(splitPoint, delimiter);

if (index < 0) {

LOG.warn("Delimiter " + Bytes.toString(delimiter) + " not found for split key "

+ Bytes.toString(splitPoint));

return splitPoint;

}

// group split keys by a prefix

return Arrays.copyOf(splitPoint, Math.min(index, splitPoint.length));

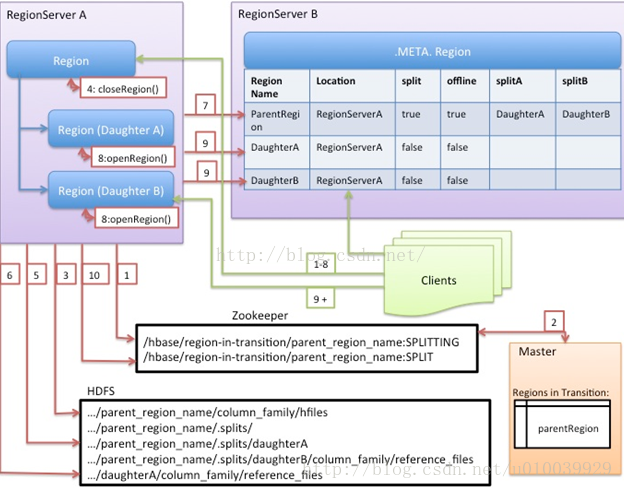
} else {

return splitPoint;

}

}

##### **HBase split的流程**



1.RegionServer触发在本地进行split，并准备split。第一步就是在zk的/hbase/region-in-transition/region-name的节点下创建一个znode节点，并置为SPLITTING状态；

2.因为Master一直watch着zk的znode，发现parentregion需要split。

3.region server 在hdfs的parent region的目录下创建一个名为“.splits”的子目录。

4.region server关闭parent region。强制flush缓存，并且在本地数据结构中标记region为下线状态。如果这个时候客户端刚好请求到parent region，会抛出NotServingRegionException。这时客户端会进行重试。

5.region server在.split目录下分别为两个daughter region（A,B）创建目录和必要的数据结构。然后创建两个引用文件指向parent regions的文件。

6.region server在HDFS中，创建真正的region目录，并且把引用文件移到对应的目录下。

7.region server发送一个put的请求到.META.表中，并且在.META.表中设置parent region为下线状态,并添加关于daughter regions的信息。但是这个时候在.META.表中daughter region 还不是独立的row，客户端在此时scan .META.表时会发现parent region在split，但是还不能获得daughter region的信息，直到她们独立的出现.META.表中。如果此时这个往.META.表中的put操作成功，parent region会高效的split，如果此时rs在RPC请求成功前失败，Master和下一个regionserver会重新打开这个parent region并将之前产生的split的脏数据清掉，.META.表成功更新后，HBase继续进行下面split的流程。

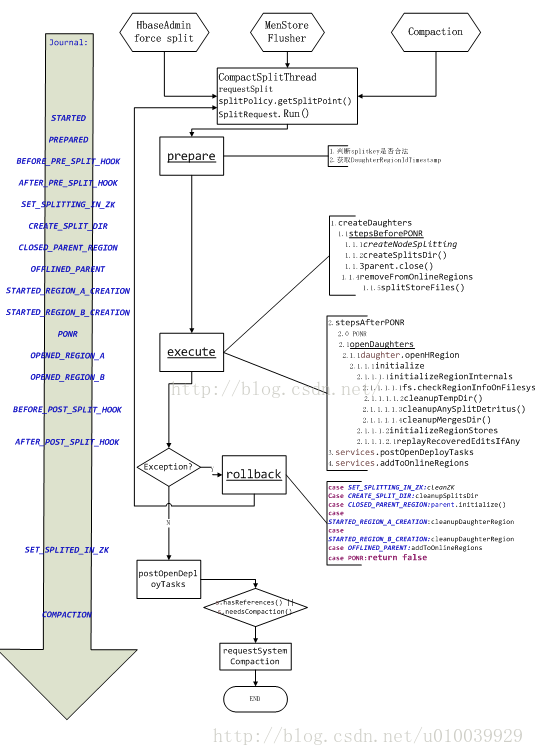
8.region server并行打开两个daughter region接受写操作。

9.region server在.META.表中增加daughters A和 B region的相关信息，在这以后，client就能发现这两个新的regions并且能发送请求到这两个新的region了。client本地具体有.META.表的缓存，当他们访问到parent region的时候，发现parent region下线了，就会重新访问.META.表获取最新的信息，并且更新本地缓存。

10.region server 更新znode 的状态为SPLIT。master就能知道状态更新了，master的平衡机制会判断是否需要把daughter regions 分配到其他region server中。

11.在split之后，meta和HDFS依然会有引用指向parentregion.当compact操作发生在daughter regions中，会重写数据file，这个时候引用就会被逐渐的去掉。GC任务会定时检测daughter regions是否还有引用指向parent files，如果没有引用指向parent files的话，parent region 就会被删除。

##### **HBase split的过程分析**



解析：

分析Hbase的split流程，先从RegionServer中的相应线程作为突破口，依次分析split的触发条件和split的执行实现。

split的触发条件：

在hbase的regionserver中维护着一个CompactSplitThread类型的变量，所有的compact/split请求都会交给该变量处理，以下是CompactSplitThread中定义的几个与split相关的变量。

private final ThreadPoolExecutor longCompactions; //long合并线程池

private final ThreadPoolExecutor shortCompactions; //short合并线程池

private final ThreadPoolExecutor splits; //split线程池

private final ThreadPoolExecutor mergePool; //merge线程池

private int regionSplitLimit;

需要注意的是最后一个整型变量regionSplitLimit，它定义了一个regionserver所持有的最大region总数，如果region的数量超过了它的限制，则split不会再发生。该变量的值由hbase.regionserver.regionSplitLimit配置，默认是1000；splits是该regionserver用于参与split的线程池，线程池中的线程数量由“hbase.regionserver.thread.split”配置，默认是1。

region发生split时，调用方会调用CompactSplitThread中的requestSplit方法，该方案将split请求包装成一个Runnable的SplitRequest对象放入前文所说的线程池split中去执行。

this.splits.execute(new SplitRequest(r, midKey, this.server));

分析requestSplit方法的调用时机就可以理出region发生split行为的时间：

第一种方式是region发生compact之后，此时会形成比较大的文件，split会在这个时候被调用；

第二种方式是region发生flush的时候，这一部分的代码如下：

lock.readLock().lock();

try {

notifyFlushRequest(region, emergencyFlush);

FlushResult flushResult = region.flush(forceFlushAllStores);

boolean shouldCompact = flushResult.isCompactionNeeded();

// We just want to check the size

boolean shouldSplit = ((HRegion)region).checkSplit() != null;

if (shouldSplit) {

this.server.compactSplitThread.requestSplit(region);

} else if (shouldCompact) {

server.compactSplitThread.requestSystemCompaction(

region, Thread.currentThread().getName());

}

if (flushResult.isFlushSucceeded()) {

long endTime = EnvironmentEdgeManager.currentTime();

server.metricsRegionServer.updateFlushTime(endTime - startTime); //将本次flush的时间记录下来

}

} catch (DroppedSnapshotException ex) {

..........

} catch (IOException ex) {

..........

} finally {

lock.readLock().unlock();

wakeUpIfBlocking(); //唤醒所有等待的线程

}

需要注意，region上的memstore发生flush的时候会获取readLock。readLock是读锁，读写锁是一种多线程同步的方案，所谓读锁其实就是共享锁，所谓写锁就是排他锁，当一个线程获取读锁时，所有其他以读模式访问的线程都可以获得访问权，而已写模式对它进行加锁的线程都会被阻塞。而当一个线程获取写锁的时候，所有其他试图获取锁的线程都会被阻塞。这里获取了readLock，所以flush时对region的更新都会被阻塞。

第三种调用是在RSRpcServices的splitRegion函数中，表示的是用户对region发出的split请求。

split的执行实现：

前面说过spilt请求会包装成SplitRequest类型的对象交由splits线程处理。所以具体的split实现是在SplitRequest的run方法中，下面我们筛选出run方法的重点流程以分析split的实现：

public void run() {

.......

SplitTransactionImpl st = new SplitTransactionImpl(parent, midKey);

try {

tableLock = server.getTableLockManager().readLock(.........);

try {

tableLock.acquire();

} catch (IOException ex) {

.......

}

if (!st.prepare()) return;

try {

st.execute(this.server, this.server);

} catch (Exception e) {

.......

}

} catch (Exception e) {

server.checkFileSystem();

} finally {

//处理post coprocessor

releaseTableLock();

//更新metrics

}

}

除了一些异常处理和回滚，run方法的主要逻辑已经梳理出来了，可以看出split前首先获取了table的共享锁，这样做的目的是防止其它并发的table修改行为修改当前table的状态或者schema等。然后初始化一个SplitTransactionImpl类型的变量，依次调用它的prepare和execute方法完成region切割。   
perpare用以完成split的前期准备，包括构造两个子HRegionInfo，分别是hri\_a和hri\_b，其中hri\_a的startKey胃parent的startKey，endKey为midKey；hri\_b的startKey为midKey，endKey为parent的endKey。

接下来在execute方法中，主要代码包括以下三步：

PairOfSameType<Region> regions = createDaughters(server, services);

if (this.parent.getCoprocessorHost() != null) {

this.parent.getCoprocessorHost().preSplitAfterPONR();

}

regions = stepsAfterPONR(server, services, regions);

transition(SplitTransactionPhase.COMPLETED);

从createDaughters方法进去，我们一点点分析split的实现，其中比较关键的步骤是createDaughters，我们把该方法的主要逻辑列出在下面：

PairOfSameType<Region> daughterRegions = stepsBeforePONR(server, services, testing);

List<Mutation> metaEntries = new ArrayList<>;

if (!testing && useZKForAssignment) {

if (metaEntries == null || metaEntries.isEmpty()) {

MetaTableAccessor.splitRegion(server.getConnection(),

parent.getRegionInfo(), daughterRegions.getFirst().getRegionInfo(),

daughterRegions.getSecond().getRegionInfo(), server.getServerName(),

parent.getTableDesc().getRegionReplication());

} else {

offlineParentInMetaAndputMetaEntries(server.getConnection(),

parent.getRegionInfo(), daughterRegions.getFirst().getRegionInfo(), daughterRegions

.getSecond().getRegionInfo(), server.getServerName(), metaEntries,

parent.getTableDesc().getRegionReplication());

}

} else if (services != null && !useZKForAssignment) {

..........

}

createDaughters返回的regions实际上是由stepsBeforePONR返回的，下面我们列出stepsBeforePONR中的主要逻辑：

this.parent.getRegionFileSystem().createSplitsDir();

try {

hstoreFilesToSplit = this.parent.close(false);

} catch (Exception e) {

.........

}

Pari<Integer,Integer> expectedReferences = splitStoreFiles(hstoreFilesToSplit);

Region a = this.parent.createDaughterRegionFromSplits(this.hri\_a);

Region b = this.parent.createDaughterRegionFromSplits(this.hri\_b);

return new PairOfSametype<Region>(a,b);

上面我们列出了几个主要步骤，穿插在这些主要步骤之间的是将当前split的状态更新到zookeeper的节点上，hbase的region在发生split的同时，会在zookeeper的region-in-transition目录下创建一个节点，供master同步监听新region的上线和老region的下线这些信息，此外如果split中间发生错误，也需要zookeeper上的状态信息同步以协调region之间的变化。

首先createSplitsDir在parent region的目录下创建了一个.split目录，接着在.split目录下创建daughter region A和region B两个子目录，然后调用close将parent关掉，调用是传入的参数false表示在关掉parent region前先强制执行一次flush，将region memstore中的数据刷写到磁盘。关闭region后region server将region标记为offline状态。

region的关闭在实现上是提交了该region拥有的store到一个线程池中，然后每个store的close方法进行关闭的，store的关闭结果异步获取。hbase在实现这一步中使用了CompletionService，返回结果通过CompletionService的take方法获取，使用这种方式的优势就是当多线程启动了多个Callable之后，每个callable都会返回一个future，CompletionService自己维护了一个线程安全的list，保证先完成的future一定先返回。

现在回来继续讲解我们的split流程，进入splitStoreFiles方法，该方法实际上是为parent中的每个storeFile创建了两个索引文件，核心代码在下面的片段中：

ThreadFactoryBuilder builder = new ThreadFactoryBuilder();

builder.setNameFormat("StoreFileSplitter-%1$d");

ThreadFactory factory = builder.build();

ThreadPoolExecutor threadPool =

(ThreadPoolExecutor) Executors.newFixedThreadPool(maxThreads, factory);

List<Future<Pair<Path,Path>>> futures = new ArrayList<Future<Pair<Path,Path>>> (nbFiles);

// Split each store file.

for (Map.Entry<byte[], List<StoreFile>> entry: hstoreFilesToSplit.entrySet()) {

for (StoreFile sf: entry.getValue()) {

StoreFileSplitter sfs = new StoreFileSplitter(entry.getKey(), sf);

futures.add(threadPool.submit(sfs)); //storefile被提交执行split了

}

}

StoreFileSpliitter最终调用HRegionFileSystem中的下面一句代码完成索引文件的创建：

Reference r =

top ? Reference.createTopReference(splitRow): Reference.createBottomReference(splitRow);

回到stepsBeforePONR方法，最后两步根据子region的元信息创建了HRegion A和B，实际上就是创建了A/B的实际存储目录。完成这些后stepsBeforePONR方法返回，然后调用如下的代码修改meta表中parent region和分裂出的region A和region B的信息。

MetaTableAccessor.splitRegion

splitRegion是一个原子方法，它将父region的信息置为offline，并写入子region的信息，但是此时的子region A和B还不能对外提供服务。需要等待regionServer打开该子region才可以，带着这个疑问，我们进入split流程的最后一个方法——stepsAfterPORN，在该方法中调用openDaughters将子RegionA和RegionB打开以接受写请求，regionsrver打开A和B之后会补充上述两个子region在.META.表中的信息，此时客户端才能够发现两个子region并向该两个region发送请求。

负责open region的线程是继承了HasThread接口的DaughterOpener类，主要包括了下面两个步骤：

1> 调用openHRegion函数的initilize，主要步骤如下：

a、向hdfs上写入.regionInfo文件以便meta挂掉时恢复；

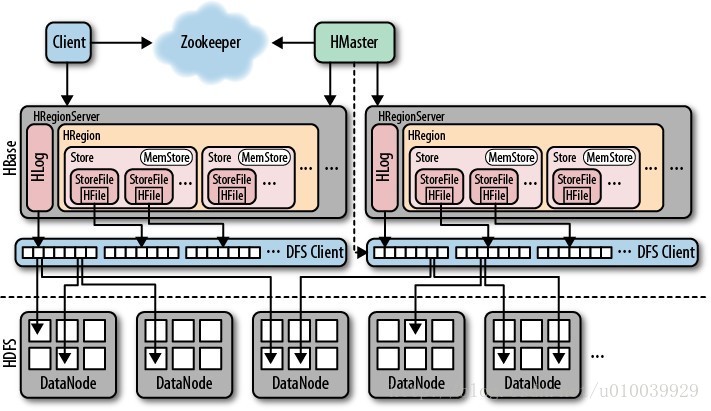
b、初始化其下的HStore，主要是调LoadStoreFiles函数；

2> 将子Region添加到rs的online region列表上，并添加到meta表中；   
最后更新zk节点上/hbase/region-in-transition/region-name节点的状态为COMPLETED，指示split结束。

Split过程结束后，HDFS和META中还会保留着指向parent region索引文件的信息，这些索引文件会在子region执行major compact重写的时候被删除掉。master的Garbage Collection任务会周期性地检查子region中是否还包含指向parents Region的索引文件，如果不再包含，此时master会将parent Region删除掉。

#### HBase Region定位原理

在[**Hbase**](http://lib.csdn.net/base/hbase)中，大部分的操作都是在RegionServer完成的，Client进行插入，删除，查询数据都需要先找到相应的RegionServer。什么叫相应的RegionServer？就是管理你要操作的那个Region的RegionServer。Client本身并不知道哪个RegionServer管理哪个Region，那么它是如何找到相应的RegionServer的？



如上图所示为[**hbase**](http://lib.csdn.net/base/hbase)的存储[**架构**](http://lib.csdn.net/base/architecture)，直观呈现了HBase的存储，在此基础上我们引入两个特殊的概念**：-ROOT-**和**.META.。**这是什么？它们是HBase的两张内置表，从存储结构和操作方法的角度来说，它们和其他HBase的表没有任何区别，你可以认为这就是两张普通的表，对于普通表的操作对它们都适用。它们与众不同的地方是HBase用它们来存贮一个重要的系统信息——Region的分布情况以及每个Region的详细信息。

好了，既然我们前面说到-ROOT-和.META.可以被看作是两张普通的表，那么它们和其他表一样就应该有自己的表结构。没错，它们有自己的表结构，并且这两张表的表结构是相同的，在分析源码之后我将这个表结构大致的画了出来：

##### -ROOT-和.META.表结构



我们来仔细分析一下这个结构，每条Row记录了一个Region的信息。

首先是RowKey，RowKey由三部分组成：TableName, StartKey 和 TimeStamp。RowKey存储的内容我们又称之为Region的Name。用来存放Region的文件夹的名字是RegionName的Hash值，因为RegionName可能包含某些非法字符。现在你应该知道为什么RegionName会包含非法字符了吧，因为StartKey是被允许包含任何值的。将组成RowKey的三个部分用逗号连接就构成了整个RowKey，这里TimeStamp使用十进制的数字字符串来表示的。这里有一个RowKey的例子：

Table1,RK10000,12345678

* 1

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

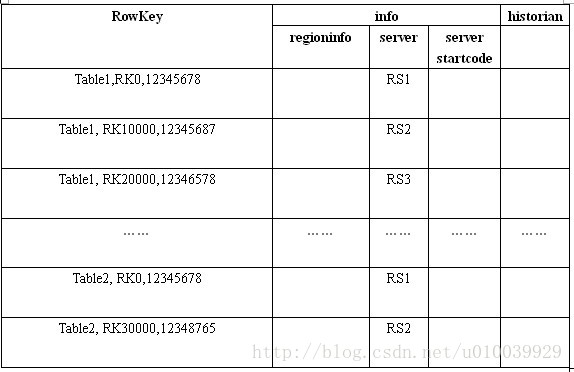
* 1

然后是表中最主要的Family：info，info里面包含三个Column：regioninfo, server, serverstartcode。其中regioninfo就是Region的详细信息，包括StartKey, EndKey 以及每个Family的信息等等。server存储的就是管理这个Region的RegionServer的地址。所以当Region被拆分、合并或者重新分配的时候，都需要来修改这张表的内容。

到目前为止我们已经学习了必须的背景知识，下面我们要正式开始介绍Client端寻找RegionServer的整个过程。我打算用一个假想的例子来学习这个过程，因此我先构建了假想的-ROOT-表和.META.表。

我们先来看.META.表，假设HBase中只有两张用户表：Table1和Table2，Table1非常大，被划分成了很多Region，因此在.META.表中有很多条Row用来记录这些Region。而Table2很小，只是被划分成了两个Region，因此在.META.中只有两条Row用来记录。这个表的内容看上去是这个样子的：

##### **.META.行记录结构**



现在假设我们要从Table2里面查询一条RowKey是RK10000的数据。那么我们应该遵循以下步骤：

1. 从.META.表里面查询哪个Region包含这条数据。
2. 获取管理这个Region的RegionServer地址。
3. 连接这个RegionServer, 查到这条数据。

好，我们先来第一步。问题是.META.也是一张普通的表，我们需要先知道哪个RegionServer管理了.META.表，怎么办？有一个方法，我们把管理.META.表的RegionServer的地址放到ZooKeeper上面不就行了，这样大家都知道了谁在管理.META.。

貌似问题解决了，但对于这个例子我们遇到了一个新问题。因为Table1实在太大了，它的Region实在太多了，.META.为了存储这些Region信息，花费了大量的空间，自己也需要划分成多个Region。这就意味着可能有多个RegionServer在管理.META.。怎么办？在ZooKeeper里面存储所有管理.META.的RegionServer地址让Client自己去遍历？HBase并不是这么做的。

HBase的做法是用另外一个表来记录.META.的Region信息，就和.META.记录用户表的Region信息一模一样。这个表就是-ROOT-表。这也解释了为什么-ROOT-和.META.拥有相同的表结构，因为他们的原理是一模一样的。

假设.META.表被分成了两个Region，那么-ROOT-的内容看上去大概是这个样子的：

##### **-ROOT-行记录结构：**



这么一来Client端就需要先去访问-ROOT-表。所以需要知道管理-ROOT-表的RegionServer的地址。这个地址被存在ZooKeeper中。默认的路径是：

/hbase/root-region-server

等等，如果-ROOT-表太大了，要被分成多个Region怎么办？嘿嘿，HBase认为-ROOT-表不会大到那个程度，因此-ROOT-只会有一个Region，这个Region的信息也是被存在HBase内部的。

现在让我们从头来过，我们要查询Table2中RowKey是RK10000的数据。整个路由过程的主要代码在org.apache.[**Hadoop**](http://lib.csdn.net/base/hadoop).hbase.client.HConnectionManager.TableServers中：

private HRegionLocation locateRegion(final byte[] tableName,

final byte[] row, boolean useCache) throws IOException {

if (tableName == null || tableName.length == 0) {

throw new IllegalArgumentException("table name cannot be null or zero length");

}

if (Bytes.equals(tableName, ROOT\_TABLE\_NAME)) {

synchronized (rootRegionLock) {

// This block guards against two threads trying to find the root

// region at the same time. One will go do the find while the

// second waits. The second thread will not do find.

if (!useCache || rootRegionLocation == null) {

this.rootRegionLocation = locateRootRegion();

}

return this.rootRegionLocation;

}

} else if (Bytes.equals(tableName, META\_TABLE\_NAME)) {

return locateRegionInMeta(ROOT\_TABLE\_NAME, tableName, row, useCache, metaRegionLock);

} else {

// Region not in the cache – have to go to the meta RS

return locateRegionInMeta(META\_TABLE\_NAME, tableName, row, useCache, userRegionLock);

}

}

这是一个递归调用的过程：

获取Table2，RowKey为RK10000的RegionServer

=> 获取.META.，RowKey为Table2,RK10000的RegionServer

=> 获取-ROOT-，RowKey为.META.,Table2,RK10000,的RegionServer

=> 获取-ROOT-的RegionServer

=> 从ZooKeeper得到-ROOT-的RegionServer

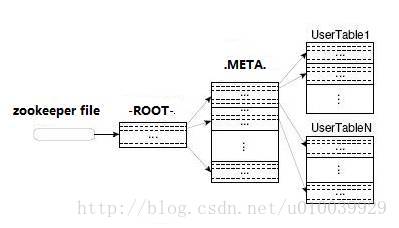
=> 从-ROOT-表中查到RowKey最接近（小于） .META.,Table2,RK10000的一条Row，并得到.META.的RegionServer

=> 从.META.表中查到RowKey最接近（小于）Table2,RK10000的一条Row，并得到Table2的RegionServer

=> 从Table2中查到RK10000的Row

到此为止Client完成了路由RegionServer的整个过程，最后要提醒大家注意两件事情：

1. 在整个路由过程中并没有涉及到MasterServer，也就是说HBase日常的数据操作并不需要MasterServer，不会造成MasterServer的负担。
2. Client端并不会每次数据操作都做这整个路由过程，很多数据都会被Cache起来。至于如何Cache，则不在本文的讨论范围之内。



HBase使用三层结构来定位region：

•1、通过zk里的文件/hbase/rs得到-ROOT-表的位置,-ROOT-表只有一个region。

•2、通过-ROOT-表查找.META.表的第一个表中相应的region的位置。其实-ROOT-表是.META.表的第一个region；.META.表中的每一个region在-ROOT-表中都是一行记录。

•3、通过.META.表找到所要的用户表region的位置。用户表中的每个region在.META.表中都是一行记录。

-ROOT-表永远不会被分隔为多个region，保证了最多需要三次跳转，就能定位到任意的region。client会讲查询的位置信息保存缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行6次网络来回，才能定位到正确的region，其中蚕丝用来发现缓存失效，另外三次用来获取位置信息。

#### HBase Flush

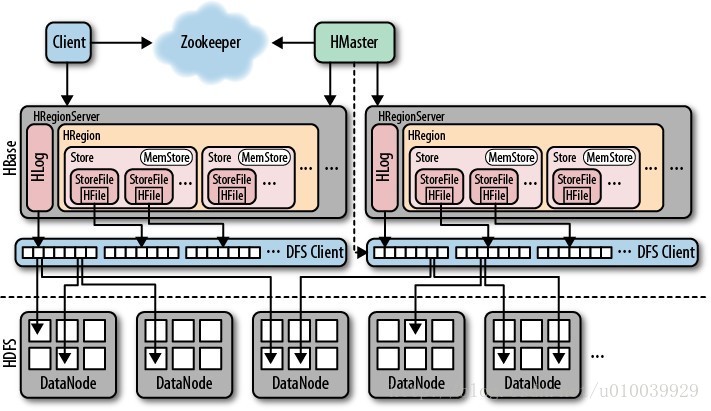
在对[Hbase](http://lib.csdn.net/base/hbase)操作中，数据读取/写入都是发生在某个HRegion下某个Store里的files。那么究竟在写入[hbase](http://lib.csdn.net/base/hbase)时，一个region下到底发生了什么呢?   
常见的有以下三种情况：

1)、memstore flush to disk

2)、columnfamily’s files compaction

3)、region split

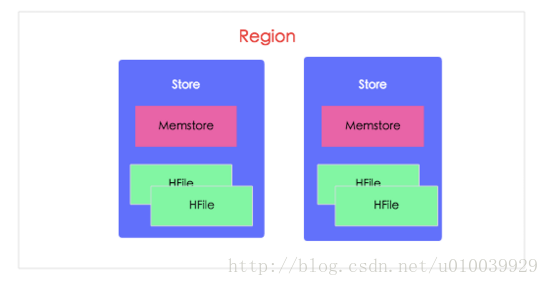
首先介绍一些概念:hbase一个表(table)会分割为n个region(在建表时可以指定多少个以及每个表的key range，同时也会在运行时split)，这些region会均匀分布在集群的regionserver上。一个region(HRegion)下会有一定数量的column family(一个cf称为一个Store，包含一个MemStore)，hbase是按列存储，所以column family是其hdfs对应的最细粒度的文件夹，文件夹的名字即是cf的名字，里面躺着一定数量的hfile(称为StoreFile)。如下图所示：



Memstore是HBase框架中非常重要的组成部分之一，是HBase能够实现高性能随机读写至关重要的一环。深入理解Memstore的工作原理、运行机制以及相关配置，对hbase集群管理、性能调优都有着非常重要的帮助。

Memstore 概述

HBase中，Region是集群节点上最小的数据服务单元，用户数据表由一个或多个Region组成。在Region中每个ColumnFamily的数据组成一个Store。每个Store由一个Memstore和多个HFile组成，如下图所示：



HBase是基于LSM-Tree模型的，所有的数据更新插入操作都首先写入Memstore中（同时会顺序写到日志HLog中），达到指定大小之后再将这些修改操作批量写入磁盘，生成一个新的HFile文件，这种设计可以极大地提升HBase的写入性能；另外，HBase为了方便按照RowKey进行检索，要求HFile中数据都按照RowKey进行排序，Memstore数据在flush为HFile之前会进行一次排序，将数据有序化；还有，根据局部性原理，新写入的数据会更大概率被读取，因此HBase在读取数据的时候首先检查请求的数据是否在Memstore，写缓存未命中的话再到读缓存中查找，读缓存还未命中才会到HFile文件中查找，最终返回merged的一个结果给用户。

可见，Memstore无论是对HBase的写入性能还是读取性能都至关重要。其中flush操作又是Memstore最核心的操作，接下来重点针对Memstore的flush操作进行深入地解析：首先分析HBase在哪些场景下会触发flush，然后结合源代码分析整个flush的操作流程，最后再重点整理总结和flush相关的配置参数，这些参数对于性能调优、问题定位都非常重要。

Memstore Flush触发方式：

1) Manual调用，HRegionInterface.flushRegion，可以被用户态org.apache.[Hadoop](http://lib.csdn.net/base/hadoop).hbase.client.HBaseAdmin调用flush操作实现，该操作会直接触发HRegion的internalFlush。

2)HRegionServer的一次更新操作，使得整个内存使用超过警戒线。警戒线是globalMemStoreLimit, RS\_JVM\_HEAPSIZE \* conf.getFloat(“hbase.regionserver.global.memstore.upperLimit”)，凡是超过这个值的情况，会直接触发FlushThread，从全局的HRegion中选择一个，将其MemStore刷入hdfs，从而保证rs全局的memstore容量在可控的范围。

HBase会在如下几种情况下触发flush操作， 需要注意的是MemStore的最小flush单元是HRegion而不是单个MemStore。可想而知，如果一个HRegion中Memstore过多，每次flush的开销必然会很大，因此我们也建议在进行表设计的时候尽量减少ColumnFamily的个数。

Memstore级别限制：当Region中任意一个MemStore的大小达到了上限（hbase.hregion.memstore.flush.size，默认128MB），会触发Memstore刷新。   
Region级别限制：当Region中所有Memstore的大小总和达到了上限（hbase.hregion.memstore.block.multiplier \* hbase.hregion.memstore.flush.size，默认 2\* 128M = 256M），会触发memstore刷新。   
Region Server级别限制：当一个Region Server中所有Memstore的大小总和达到了上限（hbase.regionserver.global.memstore.upperLimit ＊ hbase\_heapsize，默认 40%的JVM内存使用量），会触发部分Memstore刷新。Flush顺序是按照Memstore由大到小执行，先Flush Memstore最大的Region，再执行次大的，直至总体Memstore内存使用量低于阈值（hbase.regionserver.global.memstore.lowerLimit ＊ hbase\_heapsize，默认 38%的JVM内存使用量）。   
当一个Region Server中HLog数量达到上限（可通过参数hbase.regionserver.max.logs配置）时，系统会选取最早的一个 HLog对应的一个或多个Region进行flush   
HBase定期刷新Memstore：默认周期为1小时，确保Memstore不会长时间没有持久化。为避免所有的MemStore在同一时间都进行flush导致的问题，定期的flush操作有20000左右的随机延时。   
手动执行flush：用户可以通过shell命令 flush ‘tablename’或者flush ‘region name’分别对一个表或者一个Region进行flush。

Memstore Flush流程

为了减少flush过程对读写的影响，HBase采用了类似于两阶段提交的方式，将整个flush过程分为三个阶段：

prepare阶段：遍历当前Region中的所有Memstore，将Memstore中当前数据集kvset做一个快照snapshot，然后再新建一个新的kvset。后期的所有写入操作都会写入新的kvset中，而整个flush阶段读操作会首先分别遍历kvset和snapshot，如果查找不到再会到HFile中查找。prepare阶段需要加一把updateLock对写请求阻塞，结束之后会释放该锁。因为此阶段没有任何费时操作，因此持锁时间很短。

flush阶段：遍历所有Memstore，将prepare阶段生成的snapshot持久化为临时文件，临时文件会统一放到目录.tmp下。这个过程因为涉及到磁盘IO操作，因此相对比较耗时。

commit阶段：遍历所有的Memstore，将flush阶段生成的临时文件移到指定的ColumnFamily目录下，针对HFile生成对应的storefile和Reader，把storefile添加到HStore的storefiles列表中，最后再清空prepare阶段生成的snapshot。

上述flush流程可以通过日志信息查看：

/\*\*\*\*\*\*\* prepare阶段 \*\*\*\*\*\*\*\*/

2016-02-04 03:32:41,516 INFO [MemStoreFlusher.1] regionserver.HRegion: Started memstore flush for sentry\_sgroup1\_data,{\xD4\x00\x00\x01|\x00\x00\x03\x82\x00\x00\x00?\x06\xDA`\x13\xCAE\xD3C\xA3:\_1\xD6\x99:\x88\x7F\xAA\_\xD6[L\xF0\x92\xA6\xFB^\xC7\xA4\xC7\xD7\x8Fv\xCAT\xD2\xAF,1452217805884.572ddf0e8cf0b11aee2273a95bd07879., current region memstore size 128.9 M

/\*\*\*\*\*\*\* flush阶段 \*\*\*\*\*\*\*\*/

2016-02-04 03:32:42,423 INFO [MemStoreFlusher.1] regionserver.DefaultStoreFlusher: Flushed, sequenceid=1726212642, memsize=128.9 M, hasBloomFilter=true, into tmp file hdfs://hbase1/hbase/data/default/sentry\_sgroup1\_data/572ddf0e8cf0b11aee2273a95bd07879/.tmp/021a430940244993a9450dccdfdcb91d

/\*\*\*\*\*\*\* commit阶段 \*\*\*\*\*\*\*\*/

2016-02-04 03:32:42,464 INFO [MemStoreFlusher.1] regionserver.HStore: Added hdfs://hbase1/hbase/data/default/sentry\_sgroup1\_data/572ddf0e8cf0b11aee2273a95bd07879/d/021a430940244993a9450dccdfdcb91d, entries=643656, sequenceid=1726212642, filesize=7.1 M

##### RS上HRegion的选择[算法](http://lib.csdn.net/base/datastructure)

步骤1：RS上的Region，按照其MemStore的容量进行排序。

步骤2：选出Region下的Store中的StoreFile的个数未达到hbase.hstore.blockingStoreFiles，并且MemStore使用最多的Region。— bestFlushableRegion

步骤3:选出Region下的MemStore使用最多的Region。— bestAnyRegion

步骤4：如果bestAnyRegion的memstore使用量超出了bestFlushableRegion的两倍，这从另外一个角度说明，虽然当前bestAnyRegion有超过blockingStoreFiles个数的文件，但是考虑到RS内存的压力，冒着被执行Compaction的风险，也选择这个Region，因为收益大。否则，直接使用bestFlushableRegion。

##### 指定的Region写入hdfs的过程

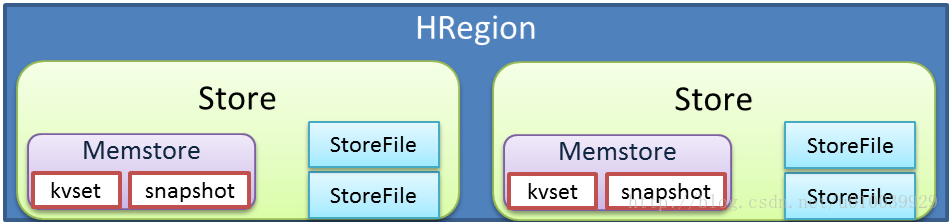
步骤1：获得updatesLock的写锁，阻塞所有对于该Region的更新操作。由此，可知Flush操作会阻塞Region区域内Row的更新操作(Put、Delete、Increment)，因为在阻塞更新操作期间，涉及到Memstore的snapshot操作，如果不做限制，那么很可能一个put操作的多个KV，分别落在kvset和snapshot当中，从而与hbase保证row的原子性相悖。

步骤2：mvcc推进一次写操作事务。每个Region维护了一个mvcc对象(Multi Version

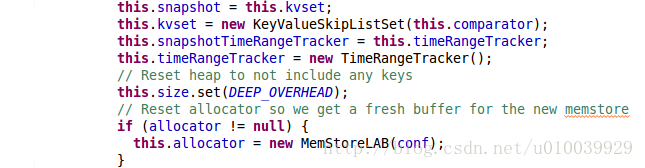
Consistency Control)，用来控制读写操作的事务性。

步骤3：从HLog中获取一个新的newSeqNum，更新HLog的lastSeqWritten。由于此时该Region的更新操作会暂停，因此，会暂时删除lastSeqWritten记录的RegionName,lastSeqNum，写入”snp”+RegionName, newSeqNum到lastSeqWritten中。这里的lastSeqWritten是HLog用来存储每个Regiond到当前时刻最后一次提交操作的SeqNum。

步骤4：为Region下的每个Store的MemStore执行snapshot操作。



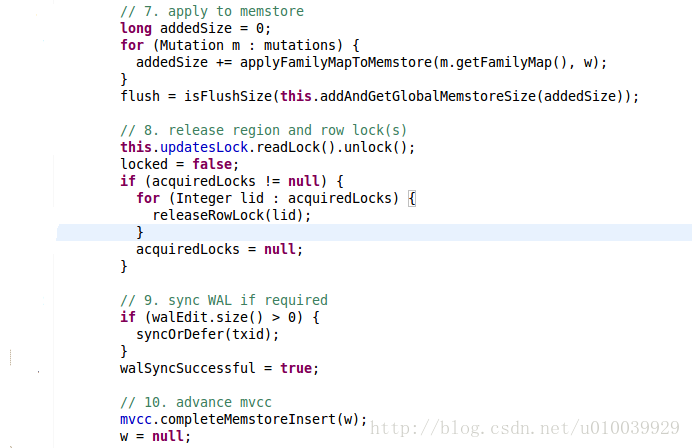
如上图所示，HRegion上Store的个数是由Table中ColumnFamily的个数确定，每个Store是由一个MemStore和数个StoreFile(HFile)文件组成，在正常的更新操作过程中，更新的内容会写入MemStore里的kvset结构中。HRegion执行Flush操作，实际上是把MemStore的内容全部刷入hdfs的过程。虽然，目前更新操作已经通过加写锁阻塞，可是读操作仍然可以继续，因此，在memstore执行snapshot的过程中，通过reference，snapshot会指向kvset，然后给kvset指向一个全新的内存区域。代码如下：



步骤5：释放updatesLock的写锁，此时该HRegion可以接收更新操作。

步骤6：更新mvcc读版本到当前写版本号。

这里有一个小插曲，在更新操作时，mvcc. completeMemstoreInsert 的操作在updatesLock的范围之外，这样在多线程高并发情况下，就存在已经写入MemStore的kvset当中，但是事务还未完成提交的情况。该场景相关代码如下：



我们可以清晰看到，通过updatesLock保证了更新操作写入了MemStore的kvset，但假定Flush线程在其它更新线之后，获得了updatesLock写锁，并执行了snapshot操作。那么，这里的mvcc就会出现读写的事务号不一致的情况，因此，在Region的Flush线程就需要使用waitForRead(w)，等待更新到目前写版本号。

步骤7：将Store内的snapshot写成一个StoreFile临时文件。

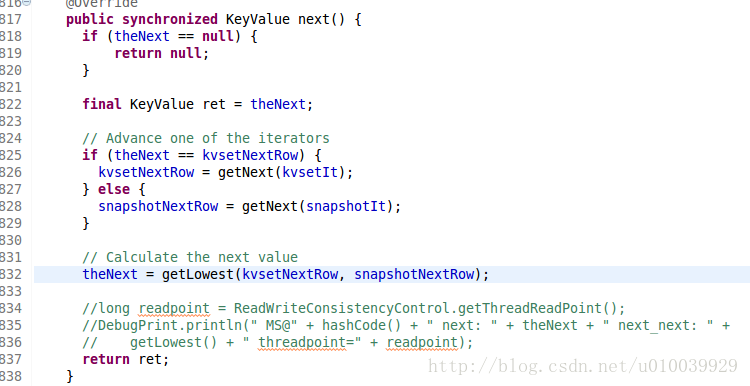
步骤8：重命名storefile文件，更新Store里文件和Memstore状态。

在步骤8完成之前，整个Hregion的读请求，是和之前没有影响的。因为在读请求过程中，StoreScanner对于kvset和snapshot进行进行同步读取，即使kvset切换成snapshot，scan的操作仍然可以继续，这部分的内容是由MemStoreScanner来控制。

在读过程中，Store里的scanner有两部份,一个是StoreFileScanner，另外一个是MemStoreScanner，它们都继承了KeyValueScanner接口，并通过StoreScanner中的KeyValueHeap封装起来。于此类似，在RegionScannerImpl也是通过一个KeyValueHeap把每个Store的StoreScanner封装起来，从而直接提供对外的服务。

读到这里，可能细心的工程师们，就会有一个疑问：Flush操作对于读的影响究竟有没有呢？

有影响，但比较小。在步骤8以前那些阶段，MemStoreScanner做到了对于kvset与snapshot的自由切换。



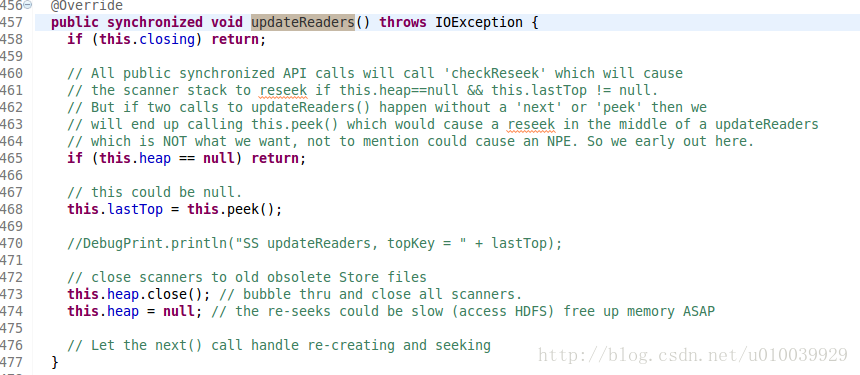
如上所示，如果kvset被重置，那么theNext将不再等于kvsetNextRow，从而切换成开始从snapshot迭代器中获取数据。

因此，在步骤1~7之间，对于读服务影响不大。但是在步骤8操作最后一步时，需要把生成storefile更新到可用的Store中的StoreFile列表，并清除snapshot的内容。

于是，此时ChangedReaderOberver就开始起作用了。

// Tell listeners of the change in readers.

notifyChangedReadersObservers();



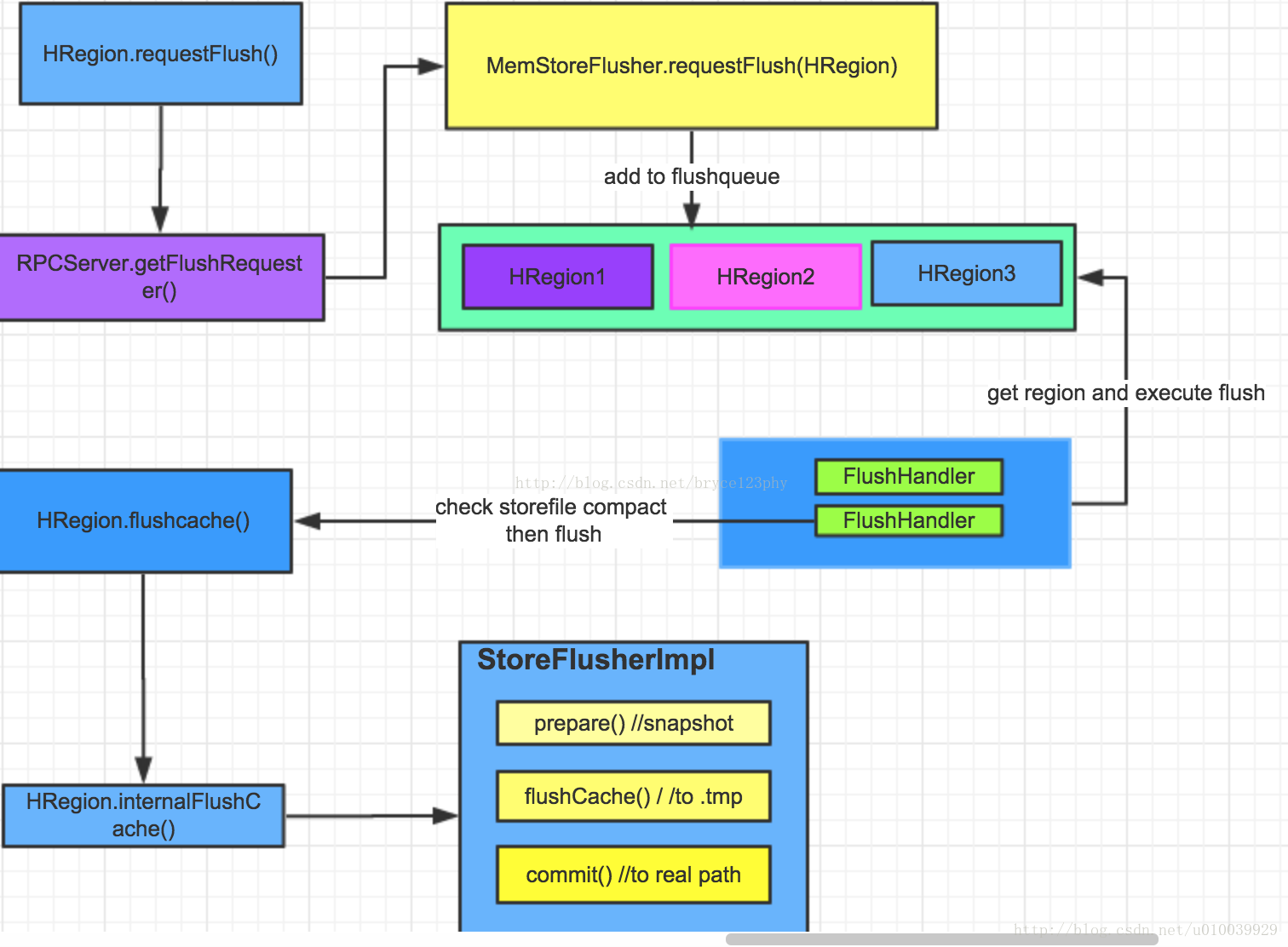
这里最为关键的是，将storescanner用来封装全部StoreFileScanner和MemStoreScanner的heap清空，它会触犯的作用是在执行next()操作时，会触发resetScannerStack操作，会重新加载Store下的所有Scanner，并执行seek到最后一次更新的key。这个过程会使得flush操作对于某些next操作变得突然顿一下。

##### MemStore flush的源码解析

flush请求的发出：

HRegion会调用requestFlush()触发flush行为，flush发生在每一处region可能发生变化的地方，包括region有新数据写入，客户端调用了put/increment/batchMutate等接口。   
首先，hbase.hregion.memstore.block.multiplier是个乘数因子，默认值是4，该值会乘上hbase.hregion.memstore.flush.size配置的值（128M），如果当前region上memstore的值大于上述两者的乘积，则该当前region的更新(update)会被阻塞住，对当前region强制发起一个flush。   
其次，还有一处要求是整个regionServer上所有memstore的大小之和是否超过了整个堆大小的40%，如果超过了则会阻塞该regionserver上的所有update，并挑选出占比较大的几个region做强制flush，直至降到lower limit以下。   
最后，当某个regionserver上的所有WAL文件数达到hbase.regionserver.max.logs（默认是32）时，该regionserver上的memstores会发生一次flush，以减少wal文件的数目，此时flush的目的是控制wal文件的个数，以保证regionserver的宕机恢复时间可控。

flush请求的处理流程：

hbase中flush请求的处理流程简化后如下图中所示，图片选自参考链接，这里逐个展开源码中的细节做介绍：   


HRegion中requestFlush()的源代码如下所示：

private void requestFlush() { //通过rsServices请求flush

if (this.rsServices == null) { //rsServices为HRegionServer提供的服务类

return;

}

synchronized (writestate) { //检查状态是为了避免重复请求

if (this.writestate.isFlushRequested()) {

return;

}

writestate.flushRequested = true; //更新writestate的状态

}

// Make request outside of synchronize block; HBASE-818.

this.rsServices.getFlushRequester().requestFlush(this, false);

if (LOG.isDebugEnabled()) {

LOG.debug("Flush requested on " + this);

}

}

关键的是下面一句：

this.rsServices.getFlushRequester().requestFlush(this, false);

其中rsServices向RegionServer发起一个RPC请求，getFlushRequester()是RegionServer中的成员变量coreFlusher中定义的方法，该变量是MemStoreFlusher类型，用于管理该RegionServer上的各种flush请求，它里面定义的几个关键变量如下：

private final BlockingQueue<FlushQueueEntry> flushQueue =

new DelayQueue<FlushQueueEntry>(); //BlockingQueue阻塞队列 DelayQueue使用优先级队列实现的无界阻塞队列

private final Map<Region, FlushRegionEntry> regionsInQueue =

new HashMap<Region, FlushRegionEntry>();

private AtomicBoolean wakeupPending = new AtomicBoolean(); //原子bool

private final long threadWakeFrequency;

private final HRegionServer server; //HRegionServer实例

private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

private final Object blockSignal = new Object(); //blockSignal定义在这里是作为一个信号量么？？？？？

protected long globalMemStoreLimit;

protected float globalMemStoreLimitLowMarkPercent;

protected long globalMemStoreLimitLowMark;

private long blockingWaitTime; //HRegion的一个阻塞更新的等待时间

private final Counter updatesBlockedMsHighWater = new Counter();

private final FlushHandler[] flushHandlers;

private List<FlushRequestListener> flushRequestListeners = new ArrayList<FlushRequestListener>(1);

下面伴随着讲解hbase的flush流程来讲解上述变量的作用。首先看requestFlush()，它将待flush的region放入待处理队列，这里包括了两个队列，flushQueue是一个无界阻塞队列，属于flush的工作队列，而regionsInQueue则用于保存位于flush队列的region的信息。

public void requestFlush(Region r, boolean forceFlushAllStores) {

synchronized (regionsInQueue) {

if (!regionsInQueue.containsKey(r)) {

// This entry has no delay so it will be added at the top of the flush

// queue. It'll come out near immediately.

FlushRegionEntry fqe = new FlushRegionEntry(r, forceFlushAllStores);

this.regionsInQueue.put(r, fqe); //将该region上的flush请求放入请求队列

this.flushQueue.add(fqe);

}

}

}

至此flush任务已经放入了工作队列，等待flush线程的处理。MemStoreFlusher中的flush工作线程定义在了flushHandler中，初始化代码如下：

int handlerCount = conf.getInt("hbase.hstore.flusher.count", 2); //用于flush的线程数

this.flushHandlers = new FlushHandler[handlerCount];

其中的handlerCount定义了regionserver中用于flush的线程数量，默认值是2，偏小，建议在实际应用中将该值调大一些。   
HRegionServer启动的时候，会一并将这些工作线程也启动，start代码如下：

synchronized void start(UncaughtExceptionHandler eh) {

ThreadFactory flusherThreadFactory = Threads.newDaemonThreadFactory(

server.getServerName().toShortString() + "-MemStoreFlusher", eh);

for (int i = 0; i < flushHandlers.length; i++) {

flushHandlers[i] = new FlushHandler("MemStoreFlusher." + i);

flusherThreadFactory.newThread(flushHandlers[i]);

flushHandlers[i].start();

}

}

接下来看看这些flusherHandler都做了什么，看看它的run方法吧，里面的主要逻辑列写在下面：

public void run() {

while (!server.isStopped()) {

FlushQueueEntry fqe = null;

try {

wakeupPending.set(false); // allow someone to wake us up again

fqe = flushQueue.poll(threadWakeFrequency, TimeUnit.MILLISECONDS);

if (fqe == null || fqe instanceof WakeupFlushThread) {

if (isAboveLowWaterMark()) {

if (!flushOneForGlobalPressure()) {

Thread.sleep(1000);

wakeUpIfBlocking();

}

wakeupFlushThread(); //wakeupFlushThread用作占位符插入到刷写队列中以确保刷写线程不会休眠

}

continue;

}

FlushRegionEntry fre = (FlushRegionEntry) fqe;

if (!flushRegion(fre)) {

break;

}

} catch (InterruptedException ex) {

continue;

} catch (ConcurrentModificationException ex) {

continue;

} catch (Exception ex) {

if (!server.checkFileSystem()) {

break;

}

}

}

synchronized (regionsInQueue) {

regionsInQueue.clear();

flushQueue.clear();

}

// Signal anyone waiting, so they see the close flag

wakeUpIfBlocking();

LOG.info(getName() + " exiting");

}

可以看到run方法中定义了一个循环，只要当前regionserver没有停止，则flusherHandler会不停地从请求队列中获取具体的请求fqe，如果当前无flush请求或者获取的flush请求是一个空请求，则根据当前regionServer上全局MemStore的大小判断一下是否需要flush。   
这里定义了两个阈值，分别是globalMemStoreLimit和globalMemStoreLimitLowMark，默认配置里前者是整个RegionServer中MemStore总大小的40%，而后者又是前者的95%，为什么要这么设置，简单来说就是，当MemStore的大小占到整个RegionServer总内存大小的40%时，该regionServer上的update操作会被阻塞住，此时MemStore中的内容强制刷盘，这是一个非常影响性能的操作，因此需要在达到前者的95%的时候，就提前启动MemStore的刷盘动作，不同的是此时的刷盘不会阻塞读写。   
回到上面的run方法，当需要强制flush的时候，调用的是flushOneForGlobalPressure执行强制flush，为了提高flush的效率，同时减少带来的阻塞时间，flushOneForGlobalPressure中对执行flush的region选择做了很多优化，总体来说，需要满足以下两个条件：   
（1）Region中的StoreFile数量不能过多，意味着挑选flush起来更快的region，减少阻塞时间；   
（2）满足条件1的所有Region中大小为最大值，意味着尽量最大化本次强制flush的执行效果；   
ok，如果请求队列中获得了flush请求，那么flush请求具体又是如何处理的呢，从代码中可以看到请求处理在flushRegion方法中，下面分析该方法都做了什么。   
它首先会检查当前region内的storeFiles的数量，如果storefile过多，会首先发出一个对该region的compact请求，然后再将region重新加入到flushQueue中等待下一次的flush请求处理，当然，再次加入到flushQueue时，其等待时间被相应缩短了。

this.flushQueue.add(fqe.requeue(this.blockingWaitTime / 100)); //将这次请求的region重新入队

storeFile数量满足要求的flush请求会进入Region的flush实现，除掉日志输出和Metrics记录，主要的代码逻辑记在下面：

private boolean flushRegion(final Region region, final boolean emergencyFlush,

boolean forceFlushAllStores) {

long startTime = 0;

synchronized (this.regionsInQueue) {

FlushRegionEntry fqe = this.regionsInQueue.remove(region);

flushQueue.remove(fqe);

} //将flush请求从请求队列中移除

lock.readLock().lock(); //region加上共享锁

try {

notifyFlushRequest(region, emergencyFlush);

FlushResult flushResult = region.flush(forceFlushAllStores);

boolean shouldCompact = flushResult.isCompactionNeeded();

boolean shouldSplit = ((HRegion)region).checkSplit() != null;

if (shouldSplit) {

this.server.compactSplitThread.requestSplit(region); //处理flush之后的可能的split

} else if (shouldCompact) {

server.compactSplitThread.requestSystemCompaction(

region, Thread.currentThread().getName()); //处理flush之后的可能compact

}

} catch (DroppedSnapshotException ex) {

server.abort("Replay of WAL required. Forcing server shutdown", ex);

return false;

} catch (IOException ex) {

if (!server.checkFileSystem()) {

return false;

}

} finally {

lock.readLock().unlock();

wakeUpIfBlocking(); //唤醒所有等待的线程

}

return true;

}

两点说明，其一是flush期间，该region是被readLock保护起来的，也就是试图获得writeLock的请求会被阻塞掉，包括move region、compact等等；其二是flush之后，可能会产生数量较多的storefile，这会触发一次compact，同样的flush后形成的较大storefile也会触发一次split；   
region.flush(forceFlushAllStores)这一句是可看出flush操作是region级别的，也就是触发flush后，该region上的所有MemStore均会参与flush，这里对region又加上了一次readLock，ReentrantReadWriteLock是可重入的，所以倒无大碍。   
该方法中还检查了region的状态，如果当前region正处于closing或者closed状态，则不会执行compact或者flush请求，这是由于类似flush这样的操作，一般比较耗时，会增加region的下线关闭时间。   
所有检查通过后，开始真正的flush实现，一层层进入调用的函数，最终的实现在internalFlushCache，代码如下：

protected FlushResult internalFlushcache(final WAL wal, final long myseqid,

final Collection<Store> storesToFlush, MonitoredTask status, boolean writeFlushWalMarker)

throws IOException {

PrepareFlushResult result

= internalPrepareFlushCache(wal, myseqid, storesToFlush, status, writeFlushWalMarker);

if (result.result == null) {

return internalFlushCacheAndCommit(wal, status, result, storesToFlush);

} else {

return result.result; // early exit due to failure from prepare stage

}

}

其中internalPrepareFlushCache进行flush前的准备工作，包括生成一次MVCC的事务ID，准备flush时所需要的缓存和中间[数据结构](http://lib.csdn.net/base/datastructure)，以及生成当前MemStore的一个快照。internalFlushCacheAndCommit则执行了具体的flush行为，包括首先将数据写入临时的tmp文件，提交一次更新事务(commit)，最后再将文件移入hdfs中的正确目录下。   
这里面我找到了几个关键点，其一，该方法是被updatesLock().writeLock()保护起来的，updatesLock与上文中提到的lock一样，都是ReentrantReadWriteLock，这里为什么还要再加锁呢。前面已经加过的锁是对region整体行为而言，如split、move、merge等宏观行为，而这里的updatesLock是数据的更新请求，快照生成期间加入updatesLock是为了保证数据一致性，快照生成后立即释放了updatesLock，保证了用户请求与快照flush到磁盘同时进行，提高系统并发的吞吐量。   
其二，那么MemStore的snapshot、flush以及commit操作具体是如何实现的，在internalPrepareFlushCache中有下面的一段代码：

for (Store s : storesToFlush) { //循环遍历该region的所有storefile,初始化storeFlushCtxs&committedFiles

totalFlushableSizeOfFlushableStores += s.getFlushableSize();

storeFlushCtxs.put(s.getFamily().getName(), s.createFlushContext(flushOpSeqId));

committedFiles.put(s.getFamily().getName(), null); // for writing stores to WAL

}

上面这段代码循环遍历region下面的storeFile，为每个storeFile生成了一个StoreFlusherImpl类，生成MemStore的快照就是调用每个StoreFlusherImpl的prepare方法生成每个storeFile的快照，至于internalFlushCacheAndCommit中的flush和commti行为也是调用了region中每个storeFile的flushCache和commit接口。   
StoreFlusherImpl中定义的flushCache主要逻辑如下：

protected List<Path> flushCache(final long logCacheFlushId, MemStoreSnapshot snapshot,

MonitoredTask status) throws IOException {

StoreFlusher flusher = storeEngine.getStoreFlusher();

IOException lastException = null;

for (int i = 0; i < flushRetriesNumber; i++) {

try {

List<Path> pathNames = flusher.flushSnapshot(snapshot, logCacheFlushId, status);

Path lastPathName = null;

try {

for (Path pathName : pathNames) {

lastPathName = pathName;

validateStoreFile(pathName);

}

return pathNames;

} catch (Exception e) {

。。。。。

}

} catch (IOException e) {

。。。。。。

}

}

throw lastException;

}

其中storeEngine是每个store上的执行引擎，flushSnapshot的目标就是将snapshot写入到一个临时目录，其实现很直观，就是使用一个InternalScanner，一边遍历cell一边写入到临时文件中。最终在commit再将tmp中的文件转移到正式目录，并添加到相应Store的文件管理器中，对用户可见。

#### [HBase compaction 分析](http://blog.csdn.net/u010039929/article/details/74216049)

Hbase为了防止小文件（被刷到磁盘的menstore）过多，保证查询效率，HBase需要在必要的时候将这些小的store file合并成相对较大的store file，这个过程就称之为compaction。在HBase中，主要存在两种类型的compaction：minor compaction和major compaction。

major compaction 的功能是将所有的store file合并成一个，触发major compaction的可能条件有：major\_compact 命令、majorCompact() API、region server自动运行（相关参数：hbase.hregion.majoucompaction 默认为24 小时、hbase.hregion.majorcompaction.jetter 默认值为0.2 防止region server 在同一时间进行major compaction）。hbase.hregion.majorcompaction.jetter参数的作用是：对参数hbase.hregion.majoucompaction 规定的值起到浮动的作用，假如两个参数都为默认值24和0,2，那么major compact最终使用的数值为：19.2~28.8 这个范围。

minor compaction的运行机制要复杂一些，它由一下几个参数共同决定：

hbase.hstore.compaction.min :默认值为 3，表示至少需要三个满足条件的store file时，minor compaction才会启动

hbase.hstore.compaction.max 默认值为10，表示一次minor compaction中最多选取10个store file

hbase.hstore.compaction.min.size 表示文件大小小于该值的store file 一定会加入到minor compaction的store file中

hbase.hstore.compaction.max.size 表示文件大小大于该值的store file 一定会被minor compaction排除

hbase.hstore.compaction.ratio 将store file 按照文件年龄排序（older to younger），minor compaction总是从older store file开始选择，如果该文件的size 小于它后面hbase.hstore.compaction.max 个store file size 之和乘以 该ratio，则该store file 也将加入到minor compaction 中。

Compaction是buffer->flush->merge的Log-Structured Merge-Tree模型的关键操作，主要起到如下几个作用：

1）合并文件

2）清除删除、过期、多余版本的数据

3）提高读写数据的效率

Minor & Major Compaction的区别

1）Minor操作只用来做部分文件的合并操作以及包括minVersion=0并且设置ttl的过期版本清理，不做任何删除数据、多版本数据的清理工作。

2）Major操作是对Region下的HStore下的所有StoreFile执行合并操作，最终的结果是整理合并出一个文件。

从这个功能上理解，Minor Compaction也不适合做Major的工作，因为部分的数据清理可能没有意义，例如，maxVersions=2，那么在少部分文件中，是否是kv仅有的2个版本也无法判断。

##### 在什么情况下会发生Compaction呢？

|  |  |  |
| --- | --- | --- |
| 参数名 | 配置项 | 默认值 |
| minFilesToCompact | hbase.hstore.compactionThreshold | 3 |
| maxFilesToCompact | hbase.hstore.compaction.max | 10 |
| maxCompactSize | hbase.hstore.compaction.max.size | Long.MAX\_VALUE |
| minCompactSize | hbase.hstore.compaction.min.size | memstoreFlushSize |

在执行压缩检查时，系统自动决定运行那种合并。在memstore被刷写到磁盘后会触发检查，或在Shell命令compact、major\_compact之后触发检查，或者是相应API在被调用后触发检查，抑或是被一个异步的后台进程触发。region服务器运行这个线程，而其功能由CompactionChecker类实现。

CompactionChecker是RS上的工作线程(Chore)，执行周期是通过threadWakeFrequency指定，大小通过Hbase.server.thread.wakefrequency配置(默认10000)，然后乘以默认倍数multiple(1000),毫秒时间转换为秒。因此，在不做参数修改的情况下，CompactionChecker大概是2hrs, 46mins, 40sec执行一次。

首先，对于HRegion里的每个HStore进行一次判断，needsCompaction()判断是否足够多的文件触发了Compaction的条件。

条件为：HStore中StoreFiles的个数 – 正在执行Compacting的文件个数 > minFilesToCompact

操作：以最低优先级提交Compaction申请。

步骤1：选出待执行Compact的storefiles。由于在Store中的文件可能已经在进行Compacting，因此，这里取出未执行Compacting的文件，将其加入到Candidates中。

步骤2:执行compactSelection算法，在Candidates中选出需要进行compact的文件，并封装成CompactSelection对象当中。

1) 选出过期的store files。过滤minVersion=0，并且storefile.maxTimeStamp + store.ttl < now\_timestamp。这意味着整个文件最大的时间戳的kv，都已经过期了，从而证明整个storefile都已经过期了。CompactSelection如果发现这样的storefile，会优先选择出来，作为Min然后提交给Store进行处理。

这部分具体操作被封装在ScanQueryMatcher下的ColumnTracker中，在StoreScanner的遍历过程，ScannerQueryMatcher负责kv的过滤。这里的ScanType包括(MAJOR\_COMPACT,MINOR\_COMPACT,USER\_SCAN)，compact操作是对选出的文件执行一次标识ScanType为MAJOR\_COMPACT或者MINOR\_COMPACT类型的scan操作，然后将最终符合标准的kv存储在一个新的文件中。

参考设置：根据应用的需求设置ttl，并且设置minVersions=0，根据selectCompation优选清理过期不保留版本的文件的策略，这样会使得这部分数据在CompactionChecker的周期内被清理。

误区：在CompactSplitThread有两个配置项

hbase.regionserver.thread.compaction.large：配置largeCompactions线程池的线程个数，默认个数为1。

hbase.regionserver.thread.compaction.small：配置smallCompactions线程池的线程个数，默认个数为1。

这两个线程池负责接收处理CR(CompactionRequest),这两个线程池不是根据CR来自于Major Compaction和Minor Compaction来进行区分，而是根据一个配置hbase.regionserver.thread.compaction.throttle的设置值(一般在hbase-site.xml没有该值的设置)，而是采用默认值2 \* minFilesToCompact \* memstoreFlushSize，如果cr需要处理的storefile文件的大小总和，大于throttle的值，则会提交到largeCompactions线程池进行处理，反之亦然。

参考设置：可以稍微调大一些largeCompactions和smallCompactions线程池内线程的个数，建议都设置成5。

2) 判断是否需要进行majorCompaction，这是很多判断条件的合成，其中最为重要的一个是   
hbase.hregion.majorcompaction设置的值，也就是判断上次进行majorCompaction到当前的时间间隔，如果超过设置值，则满足一个条件，同时另外一个条件是compactSelection.getFilesToCompact().size() < this.maxFilesToCompact。

因此，通过设置hbase.hregion.majorcompaction = 0可以关闭CompactionChecke触发的major compaction，但是无法关闭用户调用级别的mc。

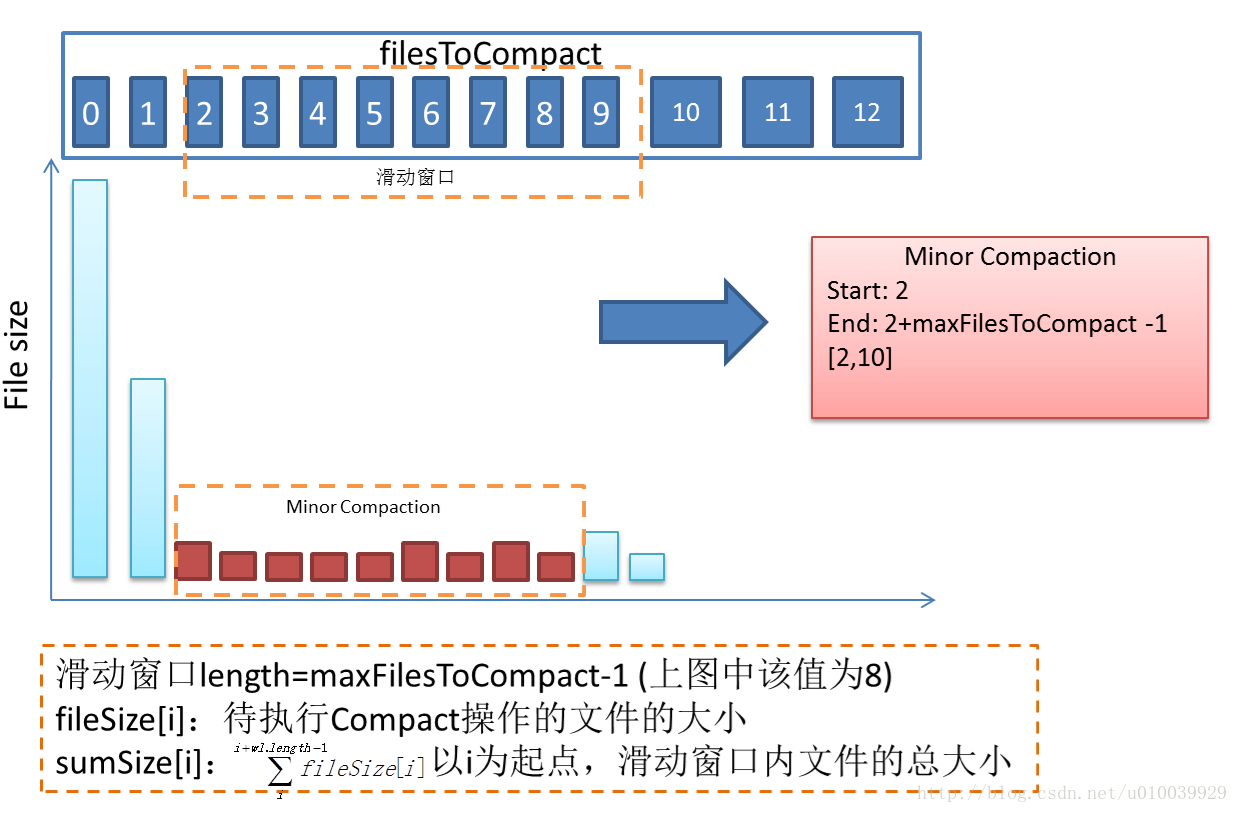
3) 过滤对于大文件进行Compaction操作。判断fileToCompact队列中的文件是否超过了maxCompactSize，如果超过，则过滤掉该文件，避免对于大文件进行compaction。

4) 如果确定Minor Compaction方式执行，会检查经过过滤过的fileToCompact的大小是否满足minFilesToCompact最低标准，如果不满足，忽略本次操作。确定执行的Minor Compaction的操作时，会使用一个smart算法，从filesToCompact当中选出匹配的storefiles。   
具体算法为：

如果fileSizes[start] > Math.max(minCompactSize, (long)(sumSize[start+1]\*r )，那么继续start++。这里r的含义是compaction比例，它有如下四个参数控制：

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 含义 |
| hbase.hstore.compaction.ratio | 1.2F |  |
| hbase.hstore.compaction.ratio.offpeak | 5.0F | 与下面两个参数联用 |
| hbase.offpeak.start.hour | -1 | 设置hbase offpeak开始时间[0,23] |
| hbase.offpeak.end.hour | -1 | 设置hbase offpeak结束时间 [0,23] |

如果默认没有设置offpeak时间的话，那么完全按照hbase.hstore.compaction.ration来进行控制。如下图所示，如果filesSize[i]过大，超过后面8个文件总和\*1.2，那么该文件被认为过大，而不纳入minor Compaction的范围。



这样做使得Compaction尽可能工作在最近刷入hdfs的小文件的合并，从而使得提高Compaction的执行效率。

5) 通过selectCompaction选出的文件，加入到filesCompacting队列中。

6) 创建compactionRequest，提交请求。

总结：

在大多数情况下，Major是发生在storefiles和filesToCompact文件个数相同，并且满足各种条件的前提下执行。这里进行几个参数配置的简介：

hbase.hregion.majorcompaction： 设置系统进行一次MajorCompaction的启动周期，如果设置为0，则系统不会主动触发MC过程。

hbase.hstore.compaction.max：设置执行Compaction(包括Major &Minor)的待合并文件的最大个数。默认值为10，如果超过该设置值，会对部分文件执行一次MinorCompaction，选择算法如Figure1。

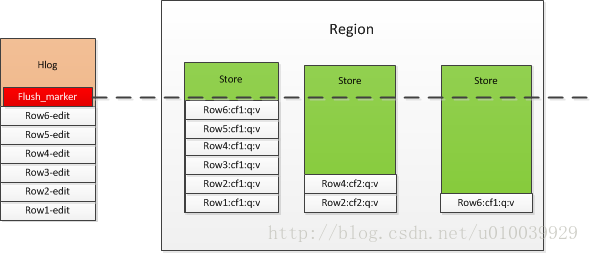
hbase.hstore.compactionThreshold: 设置执行Compaction(Major && Minor)操作的阈值，默认是3，如果想降低过频繁的合并操作，可以稍微调大一点，对于HBase负载较重的系统，可以设置成5。

##### Compaction对于读写操作的影响

Compaction与Flush不同之处在于：Flush是针对一个Region整体执行操作，而Compaction操作是针对Region上的一个Store而言，因此，从逻辑上看，Flush操作粒度较大。这属于一个LSM存储模型最核心的设计：

1）Flush操作如果只选择某个Region的Store内的MemStore写入磁盘，而不是统一写入磁盘，那么HLog上key的一致性在Reigon不同ColumnFamily(Store)下的MemStore内就会有不一致的key区间。

如下图所示，我们假定该RegionServer上仅有一个Region，由于不同的Row是在列簇上有所区别，就会出现有些不同Store内占用的内存不一致的情况，这里会根据整体内存使用的情况，或者RS使用内存的情况来决定是否执行Flush操作。如果仅仅刷入使用内存较大的memstore，那么在使用的过程中，一是Scan操作在执行时就不够统一，二是在HLog Replayer还原Region内Memstore故障前的状态，只需根据Hlog的Flush\_marker的标记位来执行Replay即可。



2）Compaction执行结束之后会生成临时文件，临时文件所在的hdfs位置如下：

/hbase-comp/comp\_cluster/ffd87a50c3df3080183d4910d183d0ee/.tmp

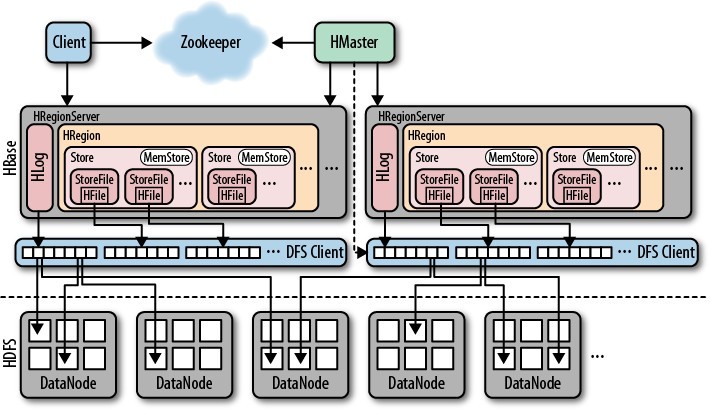
ffd87a50c3df3080183d4910d183d0ee 是comp\_cluster表格的Region名。临时文件的意义在于，在Compaction执行期间，对于原数据访问没有影响。Compaction执行合并操作生成的文件生效过程，需要对Store的写操作加锁，阻塞Store内的更新操作，直到更新Store的storeFiles完成为止。(注意，这个操作过程执行会影响到更新服务，但是影响不会太大)

3）对于读服务的影响，类似于Flush操作，也是通过ChangedReaderObserver为StoreScanner注册监听类来实现的。具体内容可以参考之前的”HBase Flush操作流程以及对读写服务的影响”。

#### HBase 存储架构

从[**Hbase**](http://lib.csdn.net/base/hbase)的[**架构**](http://lib.csdn.net/base/architecture)图上可以看出，[**hbase**](http://lib.csdn.net/base/hbase)中的存储包括HMaster、HRegionServer、HRegion、Store、MemStore、StoreFile、HFile、HLog等，本篇文章统一介绍他们的作用即存储结构。

以下是HBase存储架构图:



HBase中的每张表都通过行键按照一定的范围被分割成多个子表（HRegion），默认一个HRegion超过256M就要被分割成两个，这个过程由HRegionServer管理，而HRegion的分配由HMaster管理。

##### HMaster的作用：

* 为Region server分配region
* 负责Region server的负载均衡
* 发现失效的Region server并重新分配其上的region
* HDFS上的垃圾文件回收
* 处理schema更新请求

##### HRegionServer作用：

* 维护master分配给他的region，处理对这些region的io请求
* 负责切分正在运行过程中变的过大的region

可以看到，client访问hbase上的数据并不需要master参与（寻址访问zookeeper和region server，数据读写访问region server），master仅仅维护table和region的元数据信息（table的元数据信息保存在zookeeper上），负载很低。

HRegionServer存取一个子表时，会创建一个HRegion对象，然后对表的每个列族创建一个Store实例，每个Store都会有一个MemStore和0个或多个StoreFile与之对应，每个StoreFile都会对应一个HFile， HFile就是实际的存储文件。因此，一个HRegion有多少个列族就有多少个Store。

一个HRegionServer会有多个HRegion和一个HLog。

##### HRegion

table在行的方向上分隔为多个Region。Region是HBase中分布式存储和负载均衡的最小单元，即不同的region可以分别在不同的Region Server上，但同一个Region是不会拆分到多个server上。

Region按大小分隔，每个表一行是只有一个region。随着数据不断插入表，region不断增大，**当region的某个列族达到一个阈值**（默认256M）时就会分成两个新的region。

每个region由以下信息标识：

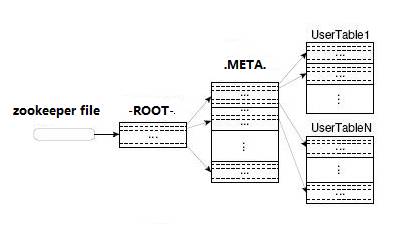
* <表名,startRowkey,创建时间>
* 由目录表(-ROOT-和.META.)可值该region的endRowkey

HRegion定位：

Region被分配给哪个Region Server是完全动态的，所以需要机制来定位Region具体在哪个region server。

HBase使用三层结构来定位region：

* 1、 通过zk里的文件/hbase/rs得到-ROOT-表的位置。-ROOT-表只有一个region。
* 2、通过-ROOT-表查找.META.表的第一个表中相应的region的位置。其实-ROOT-表是.META.表的第一个region；.META.表中的每一个region在-ROOT-表中都是一行记录。
* 3、通过.META.表找到所要的用户表region的位置。用户表中的每个region在.META.表中都是一行记录。



-ROOT-表永远不会被分隔为多个region，保证了最多需要三次跳转，就能定位到任意的region。client会讲查询的位置信息保存缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行6次网络来回，才能定位到正确的region，其中蚕丝用来发现缓存失效，另外三次用来获取位置信息。

##### Store

每一个region有一个或多个store组成，至少是一个store，hbase会把一起访问的数据放在一个store里面，即为每个ColumnFamily建一个store，如果有几个ColumnFamily，也就有几个Store。一个Store由一个memStore和0或者多个StoreFile组成。

HBase以store的大小来判断是否需要切分region。

##### MemStore

memStore 是放在内存里的。保存修改的数据即keyValues。当memStore的大小达到一个阀值（默认64MB）时，memStore会被flush到文件，即生成一个快照。目前hbase 会有一个线程来负责memStore的flush操作。

##### StoreFile

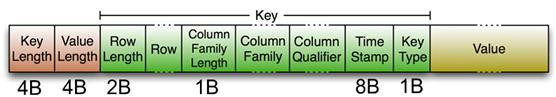
memStore内存中的数据写到文件后就是StoreFile，StoreFile底层是以HFile的格式保存。

##### HFile

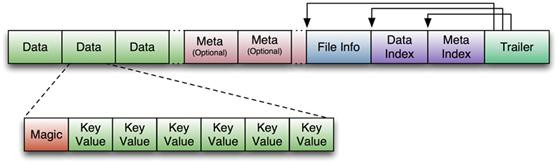
HBase中KeyValue数据的存储格式，是[**Hadoop**](http://lib.csdn.net/base/hadoop)的二进制格式文件。

首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。Trailer中又指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。

Data Block是hbase io的基本单元，为了提高效率，HRegionServer中又基于LRU的block cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定（默认块大小64KB），大号的Block有利于顺序Scan，小号的Block利于随机查询。每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成，Magic内容就是一些随机数字，目的是烦着数据损坏，结构如下。



HFile结构图如下：



Data Block段用来保存表中的数据，这部分可以被压缩。

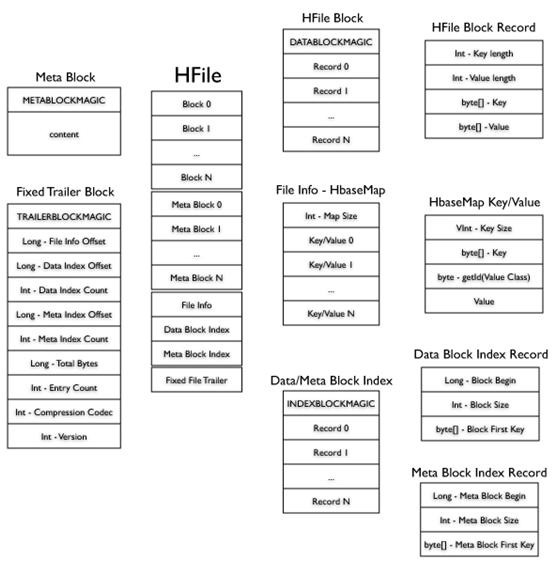
Meta Block段（可选的）用来保存用户自定义的kv段，可以被压缩。

FileInfo段用来保存HFile的元信息，本能被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index段（可选的）用来保存Meta Blcok的索引。

Trailer这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。目标HFile的压缩支持两种方式：gzip、lzo。



另外，针对目前针对现有HFile的两个主要缺陷：

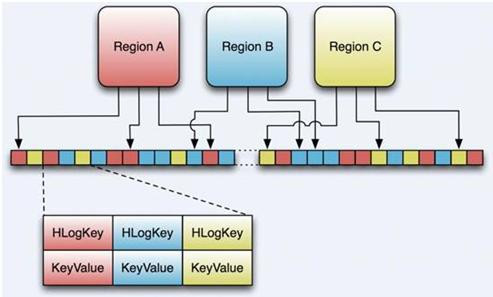
* a) 暂用过多内存
* b) 启动加载时间缓慢

提出了HFile Version2设计：<https://issues.apache.org/jira/secure/attachment/12478329/hfile_format_v2_design_draft_0.1.pdf>

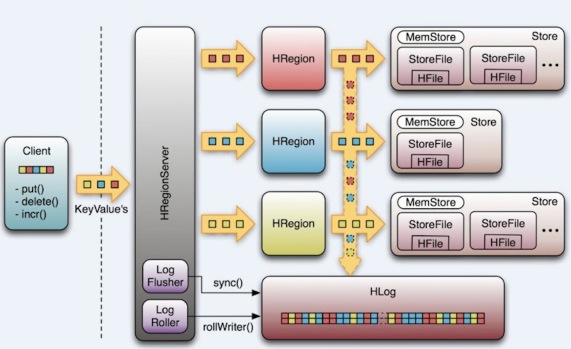
##### HLog

其实HLog文件就是一个普通的[**hadoop**](http://lib.csdn.net/base/hadoop) Sequence File， Sequence File的value是key时HLogKey对象，其中记录了写入数据的归属信息，除了table和region名字外，还同时包括sequence number和timestamp，timestamp是写入时间，equence number的起始值为0，或者是最近一次存入文件系统中的equence number。

Sequence File的value是HBase的KeyValue对象，即对应HFile中的KeyValue。



HLog(WAL log)：WAL意为write ahead log，用来做灾难恢复使用，HLog记录数据的所有变更，一旦region server 宕机，就可以从log中进行恢复。



LogFlusher

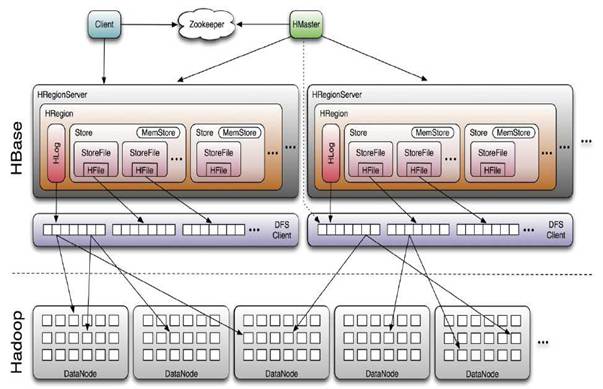
前面提到，数据以KeyValue形式到达HRegionServer，将写入WAL，之后，写入一个SequenceFile。看过去没问题，但是因为数据流在写入文件系统时，经常会缓存以提高性能。这样，有些本以为在日志文件中的数据实际在内存中。这里，我们提供了一个LogFlusher的类。它调用HLog.optionalSync(),后者根据hbase.regionserver.optionallogflushinterval (默认是10秒)，定期调用Hlog.sync()。另外，HLog.doWrite()也会根据 hbase.regionserver.flushlogentries (默认100秒)定期调用Hlog.sync()。Sync() 本身调用HLog.Writer.sync()，它由SequenceFileLogWriter实现。

LogRoller

Log的大小通过$HBASE\_HOME/conf/hbase-site.xml 的 hbase.regionserver.logroll.period 限制，默认是一个小时。所以每60分钟，会打开一个新的log文件。久而久之，会有一大堆的文件需要维护。首先，LogRoller调用HLog.rollWriter()，定时滚动日志，之后，利用HLog.cleanOldLogs()可以清除旧的日志。它首先取得存储文件中的最大的sequence number，之后检查是否存在一个log所有的条目的“sequence number”均低于这个值，如果存在，将删除这个log。

每个region server维护一个HLog，而不是每一个region一个，这样不同region（来自不同的table）的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高table的写性能。带来麻烦的时，如果一个region server下线，为了恢复其上的region，需要讲region server上的log进行拆分，然后分发到其他region server上进行恢复。

#### HBase架构组成

[**Hbase**](http://lib.csdn.net/base/hbase)采用Master/Slave[**架构**](http://lib.csdn.net/base/architecture)搭建集群，它隶属于[**Hadoop**](http://lib.csdn.net/base/hadoop)生态系统，由一下类型节点组成：HMaster节点、HRegionServer节点、ZooKeeper集群，而在底层，它将数据存储于HDFS中，因而涉及到HDFS的NameNode、DataNode等，总体结构如下：  
  
  
  
其中HMaster节点用于：

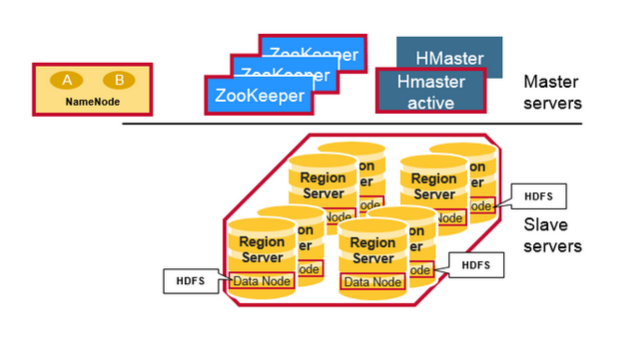
1. 管理HRegionServer，实现其负载均衡。
2. 管理和分配HRegion，比如在HRegion split时分配新的HRegion；在HRegionServer退出时迁移其内的HRegion到其他HRegionServer上。
3. 实现DDL操作（Data Definition Language，namespace和table的增删改，column familiy的增删改等）。
4. 管理namespace和table的元数据（实际存储在HDFS上）。
5. 权限控制（ACL）。

HRegionServer节点用于：

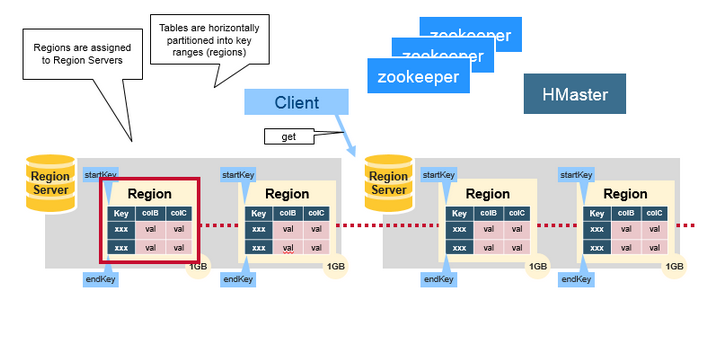
1. 存放和管理本地HRegion。
2. 读写HDFS，管理Table中的数据。
3. Client直接通过HRegionServer读写数据（从HMaster中获取元数据，找到RowKey所在的HRegion/HRegionServer后）。

ZooKeeper集群是协调系统，用于：

1. 存放整个 HBase集群的元数据以及集群的状态信息。
2. 实现HMaster主从节点的failover。

[**hbase**](http://lib.csdn.net/base/hbase) Client通过RPC方式和HMaster、HRegionServer通信；一个HRegionServer可以存放1000个HRegion；底层Table数据存储于HDFS中，而HRegion所处理的数据尽量和数据所在的DataNode在一起，实现数据的本地化；数据本地化并不是总能实现，比如在HRegion移动(如因Split)时，需要等下一次Compact才能继续回到本地化。本着半翻译的原则，再贴一个《An In-Depth Look At The HBase Architecture》的架构图：  
  
  
  
这个架构图比较清晰的表达了HMaster和NameNode都支持多个热备份，使用ZooKeeper来做协调；ZooKeeper并不是云般神秘，它一般由三台机器组成一个集群，内部使用PAXOS[**算法**](http://lib.csdn.net/base/datastructure)支持三台Server中的一台宕机，也有使用五台机器的，此时则可以支持同时两台宕机，既少于半数的宕机，然而随着机器的增加，它的性能也会下降；RegionServer和DataNode一般会放在相同的Server上实现数据的本地化。

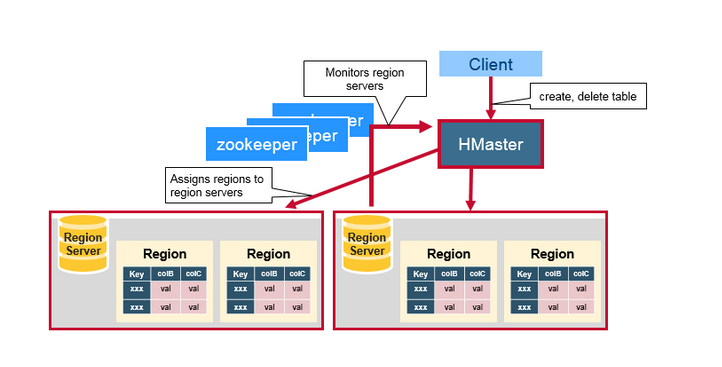
##### HRegion

HBase使用RowKey将表水平切割成多个HRegion，从HMaster的角度，每个HRegion都纪录了它的StartKey和EndKey（第一个HRegion的StartKey为空，最后一个HRegion的EndKey为空），由于RowKey是排序的，因而Client可以通过HMaster快速的定位每个RowKey在哪个HRegion中。HRegion由HMaster分配到相应的HRegionServer中，然后由HRegionServer负责HRegion的启动和管理，和Client的通信，负责数据的读(使用HDFS)。每个HRegionServer可以同时管理1000个左右的HRegion（这个数字怎么来的？没有从代码中看到限制，难道是出于经验？超过1000个会引起性能问题？来回答这个问题：感觉这个1000的数字是从BigTable的论文中来的（5   
Implementation节）：Each tablet server manages a set of tablets(typically we have somewhere between ten to a thousand tablets per tablet server)）。  
  


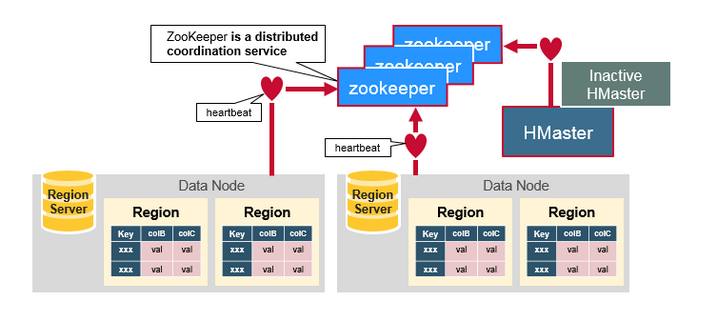
##### HMaster

HMaster没有单点故障问题，可以启动多个HMaster，通过ZooKeeper的Master Election机制保证同时只有一个HMaster出于Active状态，其他的HMaster则处于热备份状态。一般情况下会启动两个HMaster，非Active的HMaster会定期的和Active HMaster通信以获取其最新状态，从而保证它是实时更新的，因而如果启动了多个HMaster反而增加了Active   
HMaster的负担。前文已经介绍过了HMaster的主要用于HRegion的分配和管理，DDL(Data Definition Language，既Table的新建、删除、修改等)的实现等，既它主要有两方面的职责：

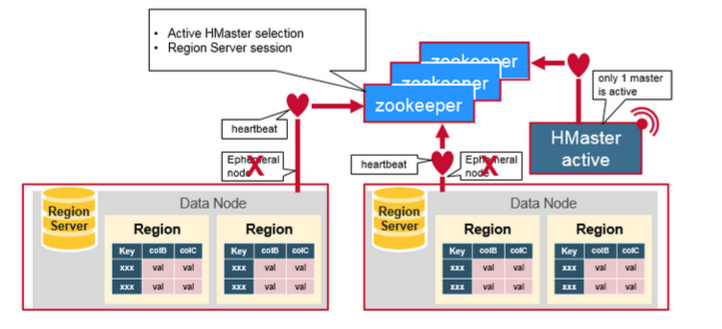
1. 协调HRegionServer
   1. 启动时HRegion的分配，以及负载均衡和修复时HRegion的重新分配。
   2. 监控集群中所有HRegionServer的状态(通过Heartbeat和监听ZooKeeper中的状态)。
2. Admin职能
   1. 创建、删除、修改Table的定义。



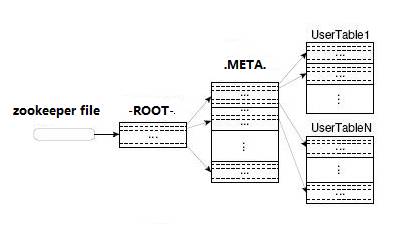
##### ZooKeeper：协调者

ZooKeeper为HBase集群提供协调服务，它管理着HMaster和HRegionServer的状态(available/alive等)，并且会在它们宕机时通知给HMaster，从而HMaster可以实现HMaster之间的failover，或对宕机的HRegionServer中的HRegion集合的修复(将它们分配给其他的HRegionServer)。ZooKeeper集群本身使用一致性协议(PAXOS协议)保证每个节点状态的一致性。  
  


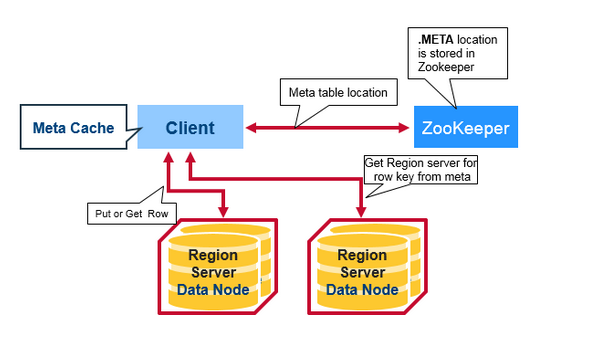
##### How The Components Work Together

ZooKeeper协调集群所有节点的共享信息，在HMaster和HRegionServer连接到ZooKeeper后创建Ephemeral节点，并使用Heartbeat机制维持这个节点的存活状态，如果某个Ephemeral节点实效，则HMaster会收到通知，并做相应的处理。  
  
  
  
另外，HMaster通过监听ZooKeeper中的Ephemeral节点(默认：/hbase/rs/\*)来监控HRegionServer的加入和宕机。在第一个HMaster连接到ZooKeeper时会创建Ephemeral节点(默认：/hbasae/master)来表示Active的HMaster，其后加进来的HMaster则监听该Ephemeral节点，如果当前Active的HMaster宕机，则该节点消失，因而其他HMaster得到通知，而将自身转换成Active的HMaster，在变为Active的HMaster之前，它会创建在/hbase/back-masters/下创建自己的Ephemeral节点。

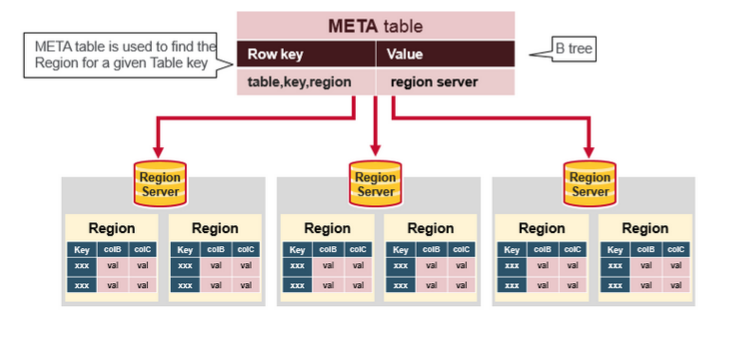
##### HBase的第一次读写

在HBase 0.96以前，HBase有两个特殊的Table：-ROOT-和.META.（如[BigTable](http://research.google.com/archive/bigtable-osdi06.pdf)中的设计），其中-ROOT- Table的位置存储在ZooKeeper，它存储了.META. Table的RegionInfo信息，并且它只能存在一个HRegion，而.META. Table则存储了用户Table的RegionInfo信息，它可以被切分成多个HRegion，因而对第一次访问用户Table时，首先从ZooKeeper中读取-ROOT- Table所在HRegionServer；然后从该HRegionServer中根据请求的TableName，RowKey读取.META. Table所在HRegionServer；最后从该HRegionServer中读取.META. Table的内容而获取此次请求需要访问的HRegion所在的位置，然后访问该HRegionSever获取请求的数据，这需要三次请求才能找到用户Table所在的位置，然后第四次请求开始获取真正的数据。当然为了提升性能，客户端会缓存-ROOT- Table位置以及-ROOT-/.META. Table的内容。如下图所示：  
  
可是即使客户端有缓存，在初始阶段需要三次请求才能直到用户Table真正所在的位置也是性能低下的，而且真的有必要支持那么多的HRegion吗？或许对Google这样的公司来说是需要的，但是对一般的集群来说好像并没有这个必要。在BigTable的论文中说，每行METADATA存储1KB左右数据，中等大小的Tablet(HRegion)在128MB左右，3层位置的Schema设计可以支持2^34个Tablet(HRegion)。即使去掉-ROOT- Table，也还可以支持2^17(131072)个HRegion， 如果每个HRegion还是128MB，那就是16TB，这个貌似不够大，但是现在的HRegion的最大大小都会设置的比较大，比如我们设置了2GB，此时支持的大小则变成了4PB，对一般的集群来说已经够了，因而在HBase 0.96以后去掉了-ROOT- Table，只剩下这个特殊的目录表叫做Meta Table(hbase:meta)，它存储了集群中所有用户HRegion的位置信息，而ZooKeeper的节点中(/hbase/meta-region-server)存储的则直接是这个Meta Table的位置，并且这个Meta Table如以前的-ROOT- Table一样是不可split的。这样，客户端在第一次访问用户Table的流程就变成了：

1. 从ZooKeeper(/hbase/meta-region-server)中获取hbase:meta的位置（HRegionServer的位置），缓存该位置信息。
2. 从HRegionServer中查询用户Table对应请求的RowKey所在的HRegionServer，缓存该位置信息。
3. 从查询到HRegionServer中读取Row。

从这个过程中，我们发现客户会缓存这些位置信息，然而第二步它只是缓存当前RowKey对应的HRegion的位置，因而如果下一个要查的RowKey不在同一个HRegion中，则需要继续查询hbase:meta所在的HRegion，然而随着时间的推移，客户端缓存的位置信息越来越多，以至于不需要再次查找hbase:meta Table的信息，除非某个HRegion因为宕机或Split被移动，此时需要重新查询并且更新缓存。  


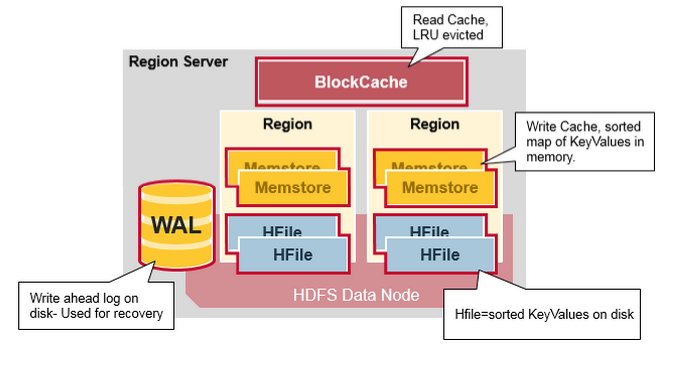
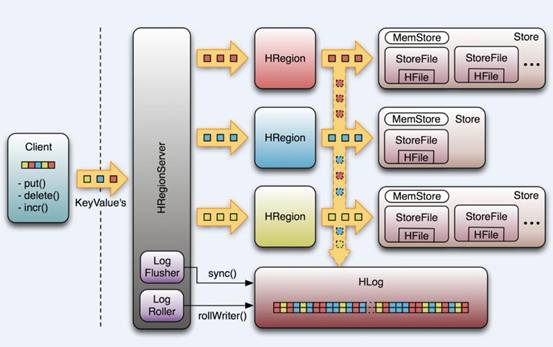
##### hbase:meta表

hbase:meta表存储了所有用户HRegion的位置信息，它的RowKey是：tableName,regionStartKey,regionId,replicaId等，它只有info列族，这个列族包含三个列，他们分别是：info:regioninfo列是RegionInfo的proto格式：regionId,tableName,startKey,endKey,offline,split,replicaId；info:server格式：HRegionServer对应的server:port；info:serverstartcode格式是HRegionServer的启动时间戳。  


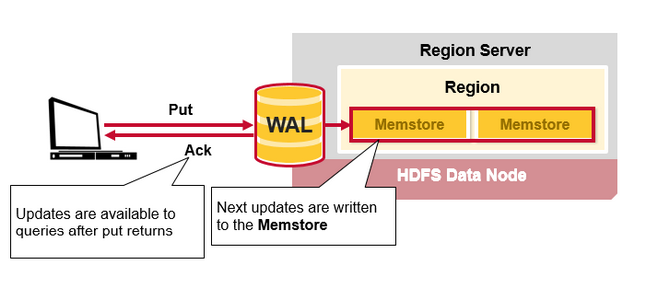
##### HRegionServer详解

HRegionServer一般和DataNode在同一台机器上运行，实现数据的本地性。HRegionServer包含多个HRegion，由WAL(HLog)、BlockCache、MemStore、HFile组成。

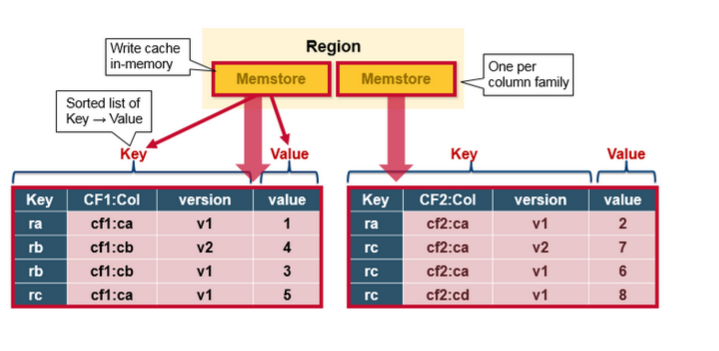
1. **WAL即Write Ahead Log**，在早期版本中称为HLog，它是HDFS上的一个文件，如其名字所表示的，所有写操作都会先保证将数据写入这个Log文件后，才会真正更新MemStore，最后写入HFile中。采用这种模式，可以保证HRegionServer宕机后，我们依然可以从该Log文件中读取数据，Replay所有的操作，而不至于数据丢失。这个Log文件会定期Roll出新的文件而删除旧的文件(那些已持久化到HFile中的Log可以删除)。WAL文件存储在/hbase/WALs/${HRegionServer\_Name}的目录中(在0.94之前，存储在/hbase/.logs/目录中)，一般一个HRegionServer只有一个WAL实例，也就是说一个HRegionServer的所有WAL写都是串行的(就像log4j的日志写也是串行的)，这当然会引起性能问题，因而在HBase 1.0之后，通过[HBASE-5699](https://issues.apache.org/jira/browse/HBASE-5699)实现了多个WAL并行写(MultiWAL)，该实现采用HDFS的多个管道写，以单个HRegion为单位。关于WAL可以参考Wikipedia的[Write-Ahead Logging](https://en.wikipedia.org/wiki/Write-ahead_logging)。顺便吐槽一句，英文版的维基百科竟然能毫无压力的正常访问了，这是某个GFW的疏忽还是以后的常态？
2. **BlockCache是一个读缓存**，即“引用局部性”原理（也应用于CPU，[分空间局部性和时间局部性](http://baike.baidu.com/link?url=Dh2u9KvowXcl2PloHJFTB9vEOoVS3WxPhDCVvbQGL_piyKFQ2iTHYAYf5mLAsFrcBUig6NC7A4-Aki61qnGCTK#3_1)，空间局部性是指CPU在某一时刻需要某个数据，那么有很大的概率在一下时刻它需要的数据在其附近；时间局部性是指某个数据在被访问过一次后，它有很大的概率在不久的将来会被再次的访问），将数据预读取到内存中，以提升读的性能。HBase中提供两种BlockCache的实现：默认on-heap LruBlockCache和BucketCache(通常是off-heap)。通常BucketCache的性能要差于LruBlockCache，然而由于GC的影响，LruBlockCache的延迟会变的不稳定，而BucketCache由于是自己管理BlockCache，而不需要GC，因而它的延迟通常比较稳定，这也是有些时候需要选用BucketCache的原因。这篇文章[BlockCache101](http://www.n10k.com/blog/blockcache-101/)对on-heap和off-heap的BlockCache做了详细的比较。
3. **HRegion是一个Table中的一个Region在一个HRegionServer中的表达**。一个Table可以有一个或多个Region，他们可以在一个相同的HRegionServer上，也可以分布在不同的HRegionServer上，一个HRegionServer可以有多个HRegion，他们分别属于不同的Table。HRegion由多个Store(HStore)构成，每个HStore对应了一个Table在这个HRegion中的一个Column Family，即每个Column Family就是一个集中的存储单元，因而最好将具有相近IO特性的Column存储在一个Column Family，以实现高效读取(数据局部性原理，可以提高缓存的命中率)。HStore是HBase中存储的核心，它实现了读写HDFS功能，一个HStore由一个MemStore 和0个或多个StoreFile组成。
   1. **MemStore是一个写缓存**(In Memory Sorted Buffer)，所有数据的写在完成WAL日志写后，会 写入MemStore中，由MemStore根据一定的算法将数据Flush到地层HDFS文件中(HFile)，通常每个HRegion中的每个 Column Family有一个自己的MemStore。
   2. **HFile(StoreFile) 用于存储HBase的数据(Cell/KeyValue)**。在HFile中的数据是按RowKey、Column Family、Column排序，对相同的Cell(即这三个值都一样)，则按timestamp倒序排列。

  
虽然上面这张图展现的是最新的HRegionServer的架构(但是并不是那么的精确)，但是我一直比较喜欢看以下这张图，即使它展现的应该是0.94以前的架构。  


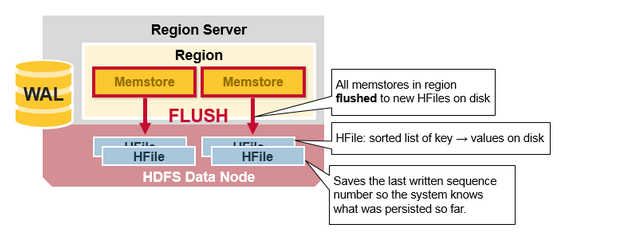
##### HRegionServer中数据写流程图解

当客户端发起一个Put请求时，首先它从hbase:meta表中查出该Put数据最终需要去的HRegionServer。然后客户端将Put请求发送给相应的HRegionServer，在HRegionServer中它首先会将该Put操作写入WAL日志文件中(Flush到磁盘中)。  
  
写完WAL日志文件后，HRegionServer根据Put中的TableName和RowKey找到对应的HRegion，并根据Column Family找到对应的HStore，并将Put写入到该HStore的MemStore中。此时写成功，并返回通知客户端。  


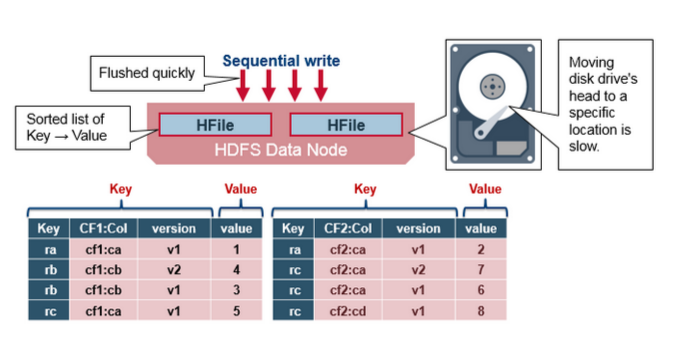
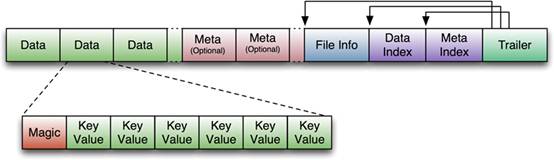
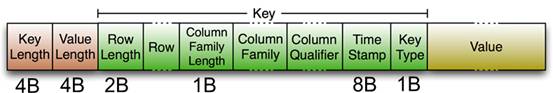
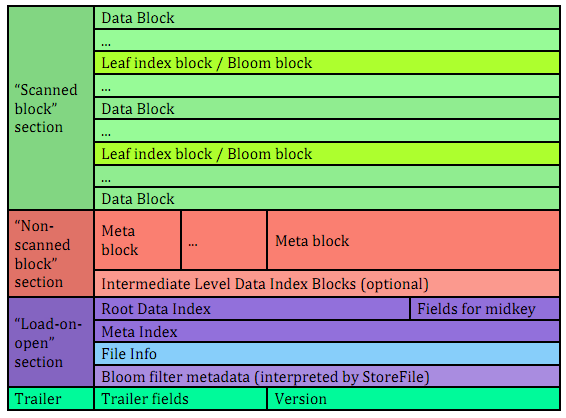
##### MemStore Flush

MemStore是一个In Memory Sorted Buffer，在每个HStore中都有一个MemStore，即它是一个HRegion的一个Column Family对应一个实例。它的排列顺序以RowKey、Column Family、Column的顺序以及Timestamp的倒序，如下所示：  
  
每一次Put/Delete请求都是先写入到MemStore中，当MemStore满后会Flush成一个新的StoreFile(底层实现是HFile)，即一个HStore(Column Family)可以有0个或多个StoreFile(HFile)。有以下三种情况可以触发MemStore的Flush动作，需要注意的是MemStore的最小Flush单元是HRegion而不是单个MemStore。据说这是Column   
Family有个数限制的其中一个原因，估计是因为太多的Column Family一起Flush会引起性能问题？具体原因有待考证。

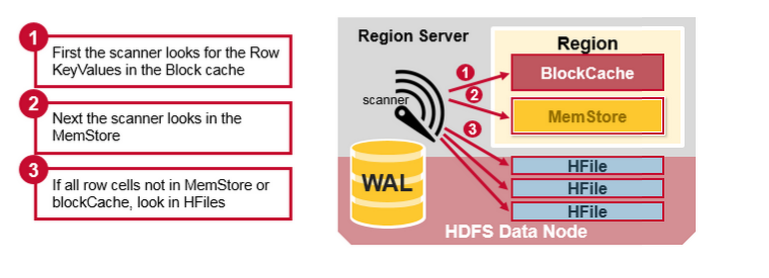
1. 当一个HRegion中的所有MemStore的大小总和超过了hbase.hregion.memstore.flush.size的大小，默认128MB。此时当前的HRegion中所有的MemStore会Flush到HDFS中。
2. 当全局MemStore的大小超过了hbase.regionserver.global.memstore.upperLimit的大小，默认40％的内存使用量。此时当前HRegionServer中所有HRegion中的MemStore都会Flush到HDFS中，Flush顺序是MemStore大小的倒序（一个HRegion中所有MemStore总和作为该HRegion的MemStore的大小还是选取最大的MemStore作为参考？有待考证），直到总体的MemStore使用量低于hbase.regionserver.global.memstore.lowerLimit，默认38%的内存使用量。
3. 当前HRegionServer中WAL的大小超过了hbase.regionserver.hlog.blocksize \* hbase.regionserver.max.logs的数量，当前HRegionServer中所有HRegion中的MemStore都会Flush到HDFS中，Flush使用时间顺序，最早的MemStore先Flush直到WAL的数量少于hbase.regionserver.hlog.blocksize \* hbase.regionserver.max.logs。[这里](http://blog.sematext.com/2012/07/16/hbase-memstore-what-you-should-know/)说这两个相乘的默认大小是2GB，查代码，hbase.regionserver.max.logs默认值是32，而hbase.regionserver.hlog.blocksize是HDFS的默认blocksize，32MB。但不管怎么样，因为这个大小超过限制引起的Flush不是一件好事，可能引起长时间的延迟，因而这篇文章给的建议：“Hint: keep hbase.regionserver.hlog.blocksize \* hbase.regionserver.maxlogs just a bit above hbase.regionserver.global.memstore.lowerLimit \* HBASE\_HEAPSIZE.”。并且需要注意，[这里](http://hbase.apache.org/book.html#_memstore_flush)给的描述是有错的(虽然它是官方的文档)。

在MemStore Flush过程中，还会在尾部追加一些meta数据，其中就包括Flush时最大的WAL sequence值，以告诉HBase这个StoreFile写入的最新数据的序列，那么在Recover时就直到从哪里开始。在HRegion启动时，这个sequence会被读取，并取最大的作为下一次更新时的起始sequence。  
  


##### HFile格式

HBase的数据以KeyValue(Cell)的形式顺序的存储在HFile中，在MemStore的Flush过程中生成HFile，由于MemStore中存储的Cell遵循相同的排列顺序，因而Flush过程是顺序写，我们直到磁盘的顺序写性能很高，因为不需要不停的移动磁盘指针。  
  
HFile参考BigTable的SSTable和[**hadoop**](http://lib.csdn.net/base/hadoop)的[TFile](http://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/file/tfile/TFile.html)实现，从HBase开始到现在，HFile经历了三个版本，其中V2在0.92引入，V3在0.98引入。首先我们来看一下V1的格式：  
  
  
V1的HFile由多个Data Block、Meta Block、FileInfo、Data Index、Meta Index、Trailer组成，其中Data Block是HBase的最小存储单元，在前文中提到的BlockCache就是基于Data Block的缓存的。一个Data Block由一个魔数和一系列的KeyValue(Cell)组成，魔数是一个随机的数字，用于表示这是一个Data   
Block类型，以快速监测这个Data Block的格式，防止数据的破坏。Data Block的大小可以在创建Column Family时设置(HColumnDescriptor.setBlockSize())，默认值是64KB，大号的Block有利于顺序Scan，小号Block利于随机查询，因而需要权衡。Meta块是可选的，FileInfo是固定长度的块，它纪录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等。Data   
Index和Meta Index纪录了每个Data块和Meta块的其实点、未压缩时大小、Key(起始RowKey？)等。Trailer纪录了FileInfo、Data Index、Meta Index块的起始位置，Data Index和Meta Index索引的数量等。其中FileInfo和Trailer是固定长度的。  
HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：  
  
  
  
开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key   
Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。随着HFile版本迁移，KeyValue(Cell)的格式并未发生太多变化，只是在V3版本，尾部添加了一个可选的Tag数组。  
HFileV1版本的在实际使用过程中发现它占用内存多，并且Bloom File和Block Index会变的很大，而引起启动时间变长。其中每个HFile的Bloom Filter可以增长到100MB，这在查询时会引起性能问题，因为每次查询时需要加载并查询Bloom Filter，100MB的Bloom   
Filer会引起很大的延迟；另一个，Block Index在一个HRegionServer可能会增长到总共6GB，HRegionServer在启动时需要先加载所有这些Block Index，因而增加了启动时间。为了解决这些问题，在0.92版本中引入HFileV2版本：  
  
  
 在这个版本中，Block Index和Bloom Filter添加到了Data Block中间，而这种设计同时也减少了写的内存使用量；另外，为了提升启动速度，在这个版本中还引入了延迟读的功能，即在HFile真正被使用时才对其进行解析。  
FileV3版本基本和V2版本相比，并没有太大的改变，它在KeyValue(Cell)层面上添加了Tag数组的支持；并在FileInfo结构中添加了和Tag相关的两个字段。关于具体HFile格式演化介绍，可以参考[这里](http://hbase.apache.org/book.html#_hfile_format_2)。对HFileV2格式具体分析，它是一个多层的类B+树索引，采用这种设计，可以实现查找不需要读取整个文件：  
  
  
  
Data Block中的Cell都是升序排列，每个block都有它自己的Leaf-Index，每个Block的最后一个Key被放入Intermediate-Index中，Root-Index指向Intermediate-Index。在HFile的末尾还有Bloom Filter用于快速定位那么没有在某个Data   
Block中的Row；TimeRange信息用于给那些使用时间查询的参考。在HFile打开时，这些索引信息都被加载并保存在内存中，以增加以后的读取性能。

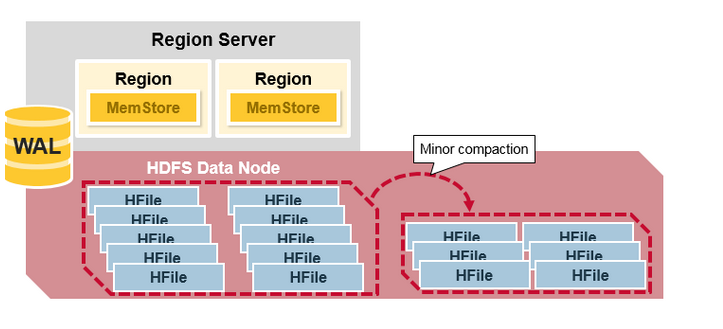
##### HBase读的实现

通过前文的描述，我们知道在[**Hbase**](http://lib.csdn.net/base/hbase)写时，相同Cell(RowKey/ColumnFamily/Column相同)并不保证在一起，甚至删除一个Cell也只是写入一个新的Cell，它含有Delete标记，而不一定将一个Cell真正删除了，因而这就引起了一个问题，如何实现读的问题？要解决这个问题，我们先来分析一下相同的Cell可能存在的位置：首先对新写入的Cell，它会存在于MemStore中；然后对之前已经Flush到HDFS中的Cell，它会存在于某个或某些StoreFile(HFile)中；最后，对刚读取过的Cell，它可能存在于BlockCache中。既然相同的Cell可能存储在三个地方，在读取的时候只需要扫瞄这三个地方，然后将结果合并即可(Merge   
Read)，在[**hbase**](http://lib.csdn.net/base/hbase)中扫瞄的顺序依次是：BlockCache、MemStore、StoreFile(HFile)。其中StoreFile的扫瞄先会使用Bloom Filter过滤那些不可能符合条件的HFile，然后使用Block Index快速定位Cell，并将其加载到BlockCache中，然后从BlockCache中读取。我们知道一个HStore可能存在多个StoreFile(HFile)，此时需要扫瞄多个HFile，如果HFile过多又是会引起性能问题。  
  


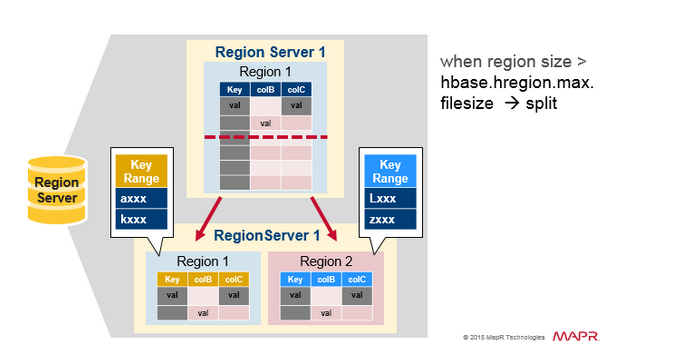
##### Compaction

MemStore每次Flush会创建新的HFile，而过多的HFile会引起读的性能问题，那么如何解决这个问题呢？HBase采用Compaction机制来解决这个问题，有点类似[**Java**](http://lib.csdn.net/base/java)中的GC机制，起初Java不停的申请内存而不释放，增加性能，然而天下没有免费的午餐，最终我们还是要在某个条件下去收集垃圾，很多时候需要Stop-The-World，这种Stop-The-World有些时候也会引起很大的问题，比如参考本人写的[这篇文章](http://www.blogjava.net/DLevin/archive/2015/08/01/426418.html)，因而设计是一种权衡，没有完美的。还是类似Java中的GC，在HBase中Compaction分为两种：Minor   
Compaction和Major Compaction。

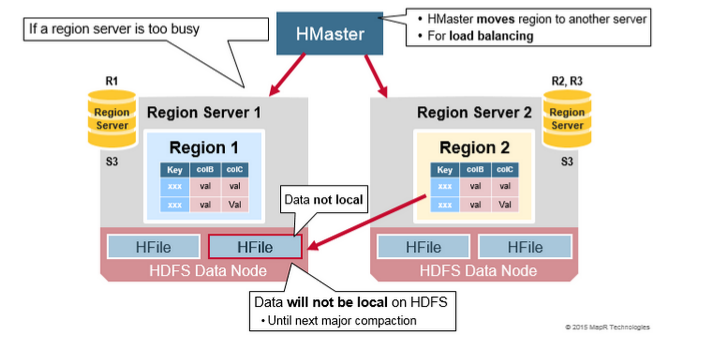
1. Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。（这个是对的吗？BigTable中是这样描述Minor Compaction的：As write operations execute, the size of the memtable in- creases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This minor compaction process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incom- ing read and write operations can continue while com- pactions occur. 也就是说它将memtable的数据flush的一个HFile/SSTable称为一次Minor Compaction）
2. Major Compaction是指将所有的StoreFile合并成一个StoreFile，在这个过程中，标记为Deleted的Cell会被删除，而那些已经Expired的Cell会被丢弃，那些已经超过最多版本数的Cell会被丢弃。一次Major Compaction的结果是一个HStore只有一个StoreFile存在。Major Compaction可以手动或自动触发，然而由于它会引起很多的IO操作而引起性能问题，因而它一般会被安排在周末、凌晨等集群比较闲的时间。

更形象一点，如下面两张图分别表示Minor Compaction和Major Compaction。  
  


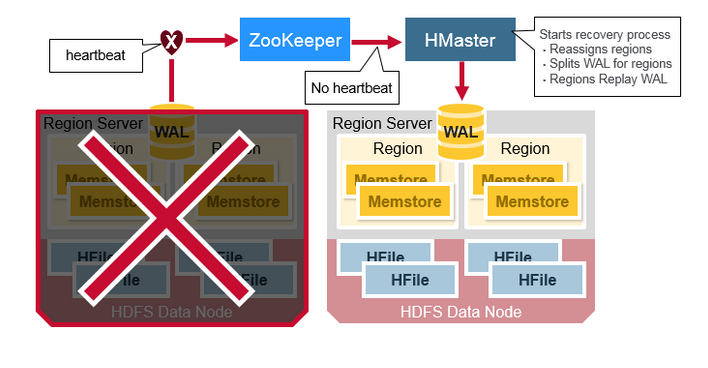
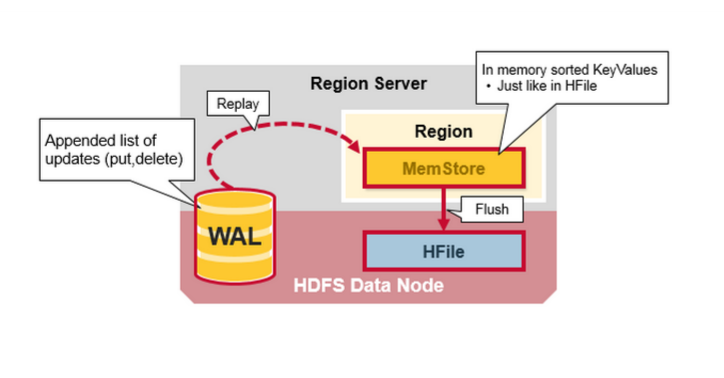
##### HRegion Split

最初，一个Table只有一个HRegion，随着数据写入增加，如果一个HRegion到达一定的大小，就需要Split成两个HRegion，这个大小由hbase.hregion.max.filesize指定，默认为10GB。当split时，两个新的HRegion会在同一个HRegionServer中创建，它们各自包含父HRegion一半的数据，当Split完成后，父HRegion会下线，而新的两个子HRegion会向HMaster注册上线，处于负载均衡的考虑，这两个新的HRegion可能会被HMaster分配到其他的HRegionServer中。关于Split的详细信息，可以参考这篇文章：[《Apache   
HBase Region Splitting and Merging》](http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/)。  
  


##### HRegion负载均衡

在HRegion Split后，两个新的HRegion最初会和之前的父HRegion在相同的HRegionServer上，出于负载均衡的考虑，HMaster可能会将其中的一个甚至两个重新分配的其他的HRegionServer中，此时会引起有些HRegionServer处理的数据在其他节点上，直到下一次Major   
Compaction将数据从远端的节点移动到本地节点。  
  


##### HRegionServer Recovery

当一台HRegionServer宕机时，由于它不再发送Heartbeat给ZooKeeper而被监测到，此时ZooKeeper会通知HMaster，HMaster会检测到哪台HRegionServer宕机，它将宕机的HRegionServer中的HRegion重新分配给其他的HRegionServer，同时HMaster会把宕机的HRegionServer相关的WAL拆分分配给相应的HRegionServer(将拆分出的WAL文件写入对应的目的HRegionServer的WAL目录中，并并写入对应的DataNode中），从而这些HRegionServer可以Replay分到的WAL来重建MemStore。  
  
  
  


#### HBase架构简单总结

在NoSQL中，存在著名的CAP理论，即Consistency、Availability、Partition Tolerance不可全得，目前市场上基本上的NoSQL都采用Partition Tolerance以实现数据得水平扩展，来处理Relational DataBase遇到的无法处理数据量太大的问题，或引起的性能问题。因而只有剩下C和A可以选择。HBase在两者之间选择了Consistency，然后使用多个HMaster以及支持HRegionServer的failure监控、ZooKeeper引入作为协调者等各种手段来解决Availability问题，然而当网络的Split-Brain(Network   
Partition)发生时，它还是无法完全解决Availability的问题。从这个角度上，Cassandra选择了A，即它在网络Split-Brain时还是能正常写，而使用其他技术来解决Consistency的问题，如读的时候触发Consistency判断和处理。这是设计上的限制。从实现上的优点：

1. HBase采用强一致性模型，在一个写返回后，保证所有的读都读到相同的数据。
2. 通过HRegion动态Split和Merge实现自动扩展，并使用HDFS提供的多个数据备份功能，实现高可用性。
3. 采用HRegionServer和DataNode运行在相同的服务器上实现数据的本地化，提升读写性能，并减少网络压力。
4. 内建HRegionServer的宕机自动恢复。采用WAL来Replay还未持久化到HDFS的数据。
5. 可以无缝的和Hadoop/MapReduce集成。

实现上的缺点：

1. WAL的Replay过程可能会很慢。
2. 灾难恢复比较复杂，也会比较慢。
3. Major Compaction会引起IO Storm。
4. 。。。。