HBase 源码分析 1-RPC 机制

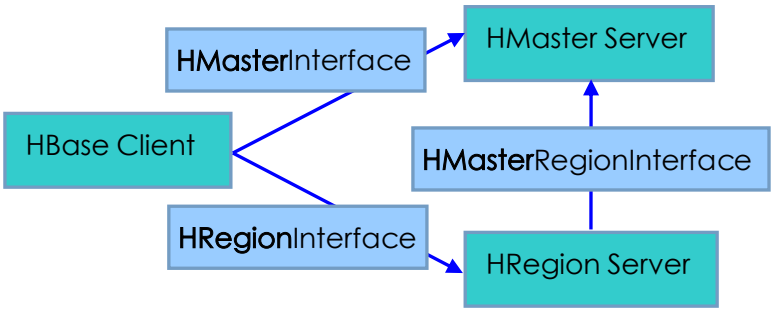
HBase 采用了和 Hadoop 相同的 RPC 机制,作为它的主要通信手段.
这是一个轻量的,不同于 Java 标准的 RMI 的一种方式.所以它的实现必须克服一些问题.如:

1) 如何分配 RPC 角色和通信信道,使得 RPC 通信可以实现.
2) 通信接口或协议的内容
3) 如何传输对象(Object),即序列化.
4) 传输,并发及会话控制
5) 其它的保障,如出错,重试等.

Question1: RPC 角色

对于第一个问题,首先要确定 RPC 通信的角色.请参看下表.

HBase 通信信道		HBase 的通信接口
客户端	服务端	
HBase Client	Master Server	HMasterInterface
HBase Client	Region Server	HRegionInterface
Region Server	Master Server	HMasterRegionInterface



HBase RPC 有明显的客户端和服务端之分.由 HBase Client,Region server(HRegionServer), Master server(HMaster)三者组成了三个信道.最右边的一列是通信两端之间约定的通信接口.客户端调用这个接口,而服务端实现这个接口. 所以最基本的工作流程就是

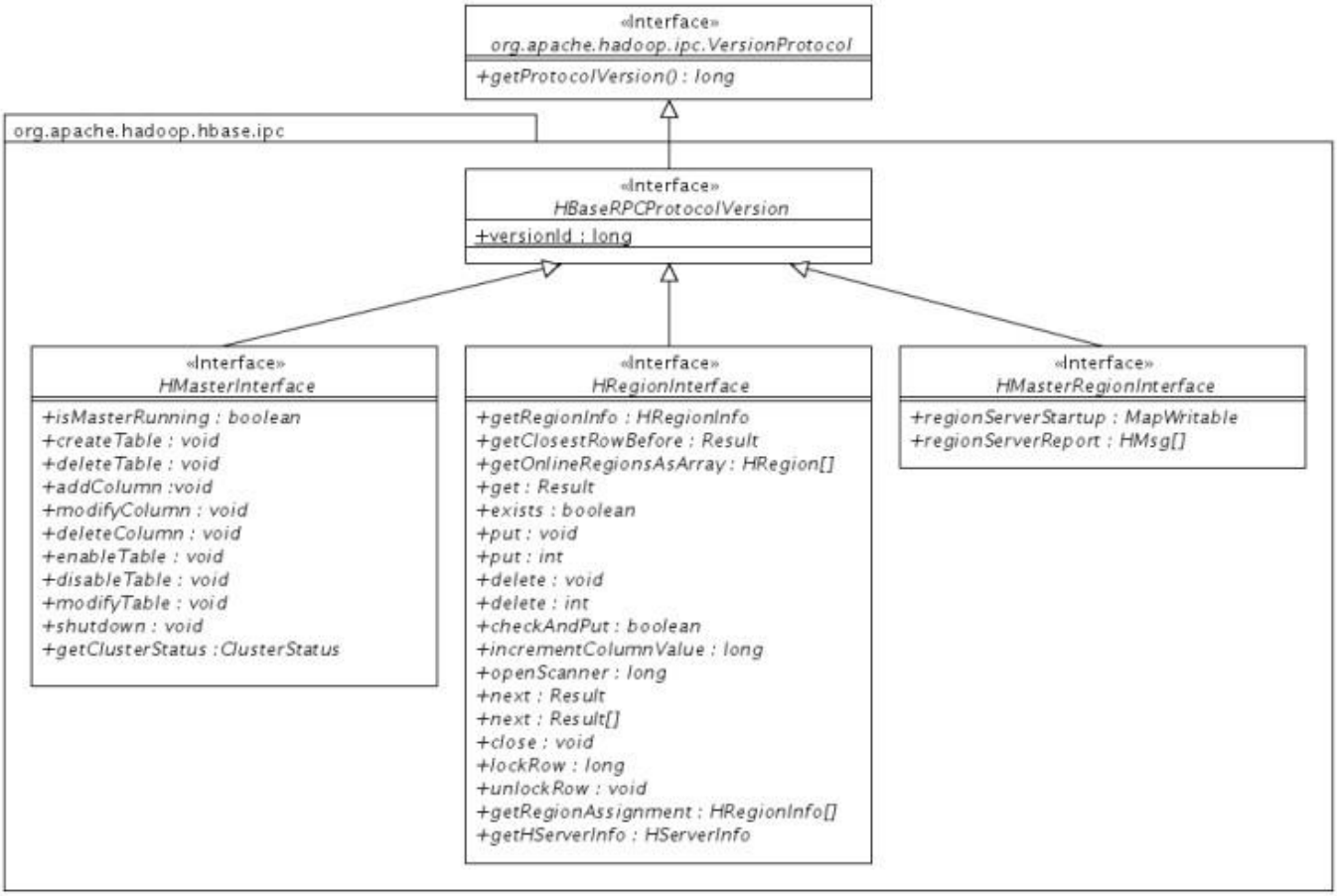
1) 客户端取得一个服务端通信接口的实例.
2) 客户端调用这个实例中的方法
3) 客户端向服务端传输调用请求
4) 服务端接口实现被调用
5) 服务端向客户端传输结果

那么除此之外的通信是不存在的吗?比如 Master 向 Region server 发出请求.答案是—否,原因很简单,一个 Master server 的实例也可以以 HBase Client 的角色来访问 Region Server,
即调用它的 HRegionInterface 接口.事实上确实如此,Master 必须调用 Region server 的接口来完成它的工作.

但是有一点可以确定的,没有 Master server 向 Region server 主动发布命令的接口,
而只有 Region server 主动向 Master server 报告和获取命令的接口,即 HMasterRegionInterface.

Question2: 通信协议

HBase 解决第二个问题,要参考三个通信接口.这三者可以说是 HBase 架构的基因



HMasterInterface

1) HMasterInterface 由 Master server(HMaster)实现, 相当于是**总管**,所以它提供的方法归纳为

a) 对**表**的增删改(CUD)的操作,及对表上线(enable),下线(disable,delete)的操作

b) 对**表的列**的增删改(CUD)操作

c) 关闭这个 HBase 集群(shutdown)和取得集群的状态(getClusterStatus)的方法.

说到底,Master server 在前两项任务上充当了 HBase Client 的角色.

只是它比较特别一点,因为它主要操作的是 Root 和 Meta 表. 这两个表是 HBase 数据架构的元数据表

对于 HBase Client 来说是如何获得这个实例呢?

请参考 o.a.h.hbase.client.HConnection.getMaster()这个方法.

HConnectionManager → HConnectionImplementation:

```
public HMasterInterface getMaster(){
  InetSocketAddress isa = new InetSocketAddress(sn.getHostname(), sn.getPort());
  rpcEngine.getProxy(HMasterInterface.class, HMasterInterface.VERSION, isa, this.conf, this.rpcTimeout);
}
```

而 HConnection 的实例是来自 o.a.h.hbase.client.HConnectionManager 的 getConnection()静态方法.

```
public HTable(Configuration conf, final byte [] tableName){
  this.connection = HConnectionManager.getConnection(conf);
}
```

HConnection 真正的实现是 HConnectionManager.~~TableServers~~ → HConnectionImplementation

```
static class HConnectionImplementation implements HConnection, Closeable {
```

HRegionInterface

2) HRegionInterface，由 Region Server(HRegionServer)实现.这是 HBase 主要的**数据操作接口**.它的功能有

a) Get 操作,读**表记录**的操作
b) Put 操作,检查及 Put 的组合操作,列值的增值的操作,即写表记录的操作
c) Delete 操作,删除表记录的操作
d) Scan 操作,分为打开 Scanner(openScanner),得到下一条或多条的记录及关闭 Scanner 的操作
e) 行锁及解锁操作
f) 定位(查找)的操作
g) 取得本 Region server 上 Region 信息,服务器信息的操作

Region server 顾名思义就是为 Region 服务的服务器.所以它的主体的方法都是针对一个 Region 的.
一个 Region 也就是一个表的横向片断,表太大了,所以要分给不同的 Region Server 来管理.
从 HBase client 的角度来说,因为一个对数据表的操作可能要跨多个 Region,也就是要访问多个 Region server 所以这个工作就必须由 client 来完成.HBase client 的 API 就是要解决这个复杂性.

HRegionInterface 的实例同样来自 HConnection 接口的实例 HConnectionManager.~~TableServers~~.

```
static class HConnectionImplementation implements HConnection, Closeable {
  private final Class<? extends HRegionInterface> serverInterfaceClass;   → HRegionInterface
  private volatile HMasterInterface master;   → HMasterInterface
}
```

获取一个 HRegionInterface 实例:

```
HRegionInterface getHRegionConnection(final String hostname, final int port, final InetSocketAddress isa...){
  HRegionInterface server = HBaseRPC.waitForProxy(
    this.rpcEngine, serverInterfaceClass, HRegionInterface.VERSION, address,...);
}
```

HMasterRegionInterface

3) HMasterRegionInterface，由 Master server 实现.它只有两个的方法,

a) 向 Master server 报告我启动了,Master server 就回给它当前的配置,如文件系统,hbase 的根目录
b) 向 Master server 报告状态,及管理的 Region 的信息,顺便从 Master server 取得要执行的命令.

有哪些命令和报告呢,参考 org.apache.hadoop.hbase.HMsg.Type.有命令：打开,关闭,切分,压缩和停止 Region;报告有 Region serve 打开,关闭,切分,压缩,更新,Region 正退出等.

值得注意的是,我们如从 CRUD(Create, Read, Update, Delete)这个角度来看这套接口,似乎是不完整的.注意 HMasterInterface,它只提供了 CUD 三个对表的操作.那么对表的读取在哪里呢?另一方面可以看到的是 HRegionInterface 指供了完整的 CRUD.所以有一个潜在的方法就是对表的读取可以通过 HRegionInterface 对 Root 表及 Meta 表的读取来完成,事实上,HBase 的客户端代码也是这样做的,它利用 scanner 扫描 Root 和 Meta 表来读取表的信息.

Question3: 序列化

第三个问题相对比较简单,可以看到它是利用 org.apache.hadoop.io.Writeable 这个接口来进行序列化的,而不是通过标准的 Serialize 接口.Writeable 接口有两个方法,分别是

1) write(DataOutput out) 将数据写入流中(序列化)
2) readFields(DataInput in) 从流中读出这数据实例化(反序列化)这个对象.

还要一个隐含的要求,就是实现这个接口的类要有公有的无参构造器.最后会说明理由.
也就是说只要实现这个接口的类的实例就可以在 HBase 的 RPC 中传输,作为函数调用的参数或返回值.

具体一些来说, org.apache.hadoop.hbase.io.HbaseObjectWritable 是真正被传输用到的,它是一个封装器,有两个静态的表,一个是类 code(一个内部分配的 Byte)到类实例的映射表，一个是类实例到类 code 的映射表.

```
public class HbaseObjectWritable implements Writable, WritableWithSize, Configurable {
  // Here we maintain two static maps of classes to code and vice versa.
  // Add new classes+codes as wanted or figure way to auto-generate these maps from the HMasterInterface.
  static final Map<Integer, Class<?>> CODE_TO_CLASS = new HashMap<Integer, Class<?>>();
  static final Map<Class<?>, Integer> CLASS_TO_CODE = new HashMap<Class<?>, Integer>();
```

在这两个表静态初始化时,将 HBase 中所有要传输的类(Writable),原型数据(int, boolean)及一些特别的类(Stirng, 数组)等都编成 code 存入这张表中.对于它的实例来说,它提供了 set,get 方法将一个对象存入或读出,然后按不同对象采用不同的方法来将数据存入流中或从流中读出.

重新归纳一下这个序列化的过程,

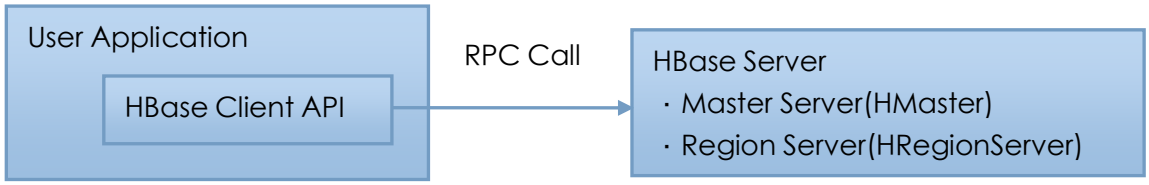
1) Client 调用 RPC 接口时,一些原型值,String,或 Writable 对象传入了前述三个接口的方法.
2) 这些参数被装入 HbaseObjectWritable 对象中.
3) HbaseObjectWritable 将这些对象的类 code(查静态表)写入流中,并将对象实例的数据写入流中.
4) Server 将接收的参数反序列化.先生成 HbaseObjectWritable 实例
5) 用这个实例从流中读到类 code,实例化.然后按它的类型读出它的数据.
6) 这样就可以调用 HbaseObjectWritable.get()方法将这个构建出的对象取出了.

另一部分返回结果的传输与上述过程类似. 请注意第 5 步,其中的实例化,对于 Writable 对象而言,必须要实现无参构造器.这样才能帮助它方便的实例化成.其实还可以有其它实例化的方法,WritableFactory 这儿就不详述了.

第四个问题和第五个问题将涉及传输的两端,将分别在客户端和服务端分别讲到.

HBase 源码分析 2-RPC 客户端

客户端 – 指的是 HBase client API.提供了从用户程序连接到 HBase 后台服务器(下面就介绍到)的功能
服务端 – 即指的是 HBase 的 Master server 及 Region server
用户端 – 指用户程序.即对 HBase client API 的调用方.

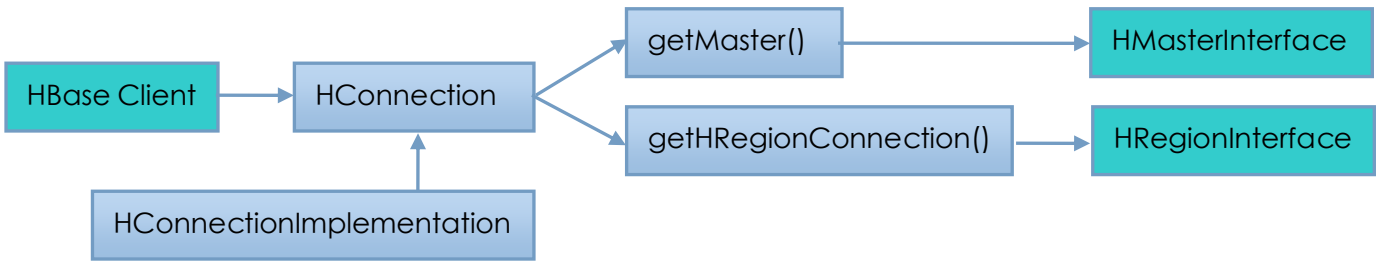


本篇的主要目的是说明 RPC 的客户端实现.解决客户端 RPC 的最后两个问题

4) 传输,并发及会话控制
5) 其它的保障,如出错,重试等.

Question4: 传输，并发，会话

前一篇基础,已经说明 HBase client 可以得到 HMasterInterface 和 HRegionInterface 的接口实例.具体参看 org.apache.hadoop.hbase.client.HConnectionManager.~~TableServers~~HConnectionImplementation 的两个方法 **getMaster()**和 **getHRegionConnection()**.



从这点说,HBase 的 RPC 与 RMI 是相似的,即通本地的桩(stub)对象,来传输调用,但对用户来说,这是透明的.
这个实现主要分两大步骤

1. 生成接口的实例.这是通过代理类来实现的.
2. 方法调用转化为序列化的 socket 传输.
3. 代理类将方法,参数序列化后,用 socket 传给服务端

Step1: 实例化

第一步,这两个接口的实例,是如何实例化出来? 毫无疑问的是这两个实例是桩(stub)或称为代理,桩的创建是从 org.apache.hadoop.hbase.ipc.HBaseRPC.getProxy()这个静态方法中得到的.

```
public <T extends VersionedProtocol> T getProxy(Class<T> protocol, long clientVersion,
      InetSocketAddress addr, Configuration conf, int rpcTi(1)ut)
   T proxy = (T) Proxy.newProxyInstance(protocol.getClassLoader(), new Class[] { protocol },(2)
         (3)new Invoker(client, protocol, addr, User.getCurrent(), conf, HBaseRPC.getRpcTimeout(rpcTimeout)));
}
```

下面就详细的说明一下 getProxy() ,它的核心是利用的 java 反射中的动态代理框架.
调用 java.lang.reflect.Proxy 的静态方法 newProxyInstance,它需要三个参数:

1) ClassLoader,
2) 要实现的接口类(可以有多个),例如 HRegionInterface,
3) 最后是函数调用代理类 java.lang.reflect.InvocationHandler 的实例.

也就是说 Proxy.newProxyInstantce 的功能是将产生一个 Object,这个 Object 实现了指定的接口,
其实就是将接口与 InvocationHandler 的实例绑定了.
对接口中方法的调用将被转发到 InvocationHandler 实例上.
getProxy()将生成的对象转化成 VersionProtocol 对象然后返回.
然后再根据外部的调用将 VersionProtocol 转化为具体的 HRegionInterface 或 HMasterInterface

在这步,InvocationHandler 是由 HBaseRPC 的内部类 Invoker 实现的.所有的 RPC 调用就落实到了它的 invoke 方法上.invoke()又利用 org.apache.hadoop.hbase.HBaseClient 来完成调用的传输.

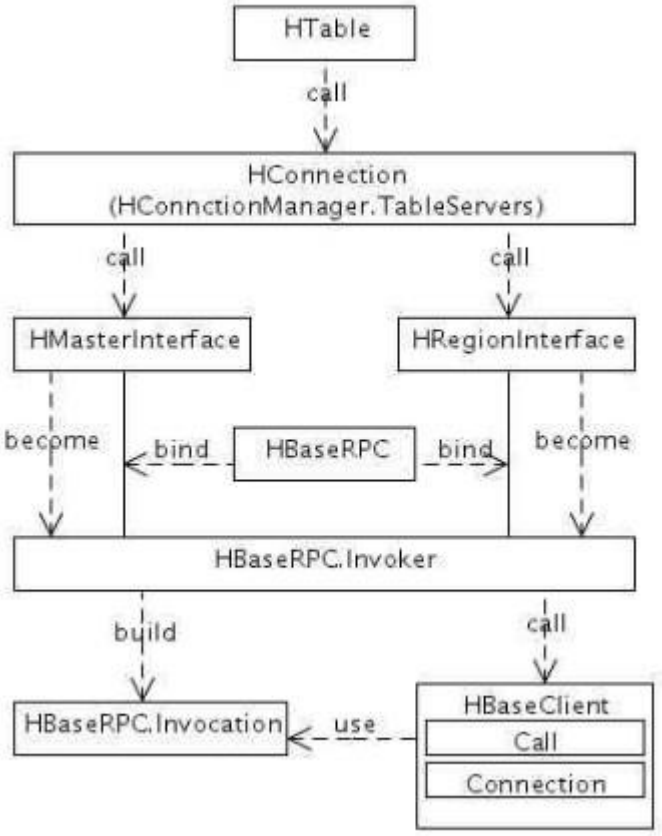
```
private static class Invoker implements InvocationHandler {
   private Class<? extends VersionedProtocol> protocol;
   private HBaseClient client;

   public Object invoke(Object proxy, Method method, Object[] args){
     HbaseObjectWritable value = (HbaseObjectWritable)
   (5)client.call(new Invocation(method, protocol, args), address,protocol, ticket, rpcTimeout);
     return value.get();    (4)
   }
```

Step2: 序列化

4) 调用的方法和参数被封装成 Invocation(~~HBaseRPC 内部类~~)对象. Invocation 本身就是一个 Writable 对象.函数名被传化成一个编码 code.可以看到 Invocation 内部有两个表,即方法名到一个字节值的双向的映射.首先方法名被排序了,然后它们对应的字节 code 从 0 依次增一.重载的方法编码相同,但可以根据参数不同来区分的.参数依次被序列化成 HbaseObjectWritable 对象.通过这个 Invocation 对象,调用就可以输出成数据流或从数据流中输入.

5) 这个协议的下层封装是由 HBaseClient 的内部类 Call 实现的,Call 装入前述的 Invocation 对象,从当前的连接类 HBaseClient 取得一个 id,这是一个自增量,然后将这个 id 及流的长度及 Invocation 通过 socket 连接发送出去.并同步等待(Call.Wait())返回结果.Call 对象实例被放入了另一个内部类 Connection 的一个表 calls 中,是 id(integer)到 Call 实例的映射.如果 Call 实例的完成标志被置,则说明结果被保存在了它的 value 字段中.后续反序列化的就不详述了.



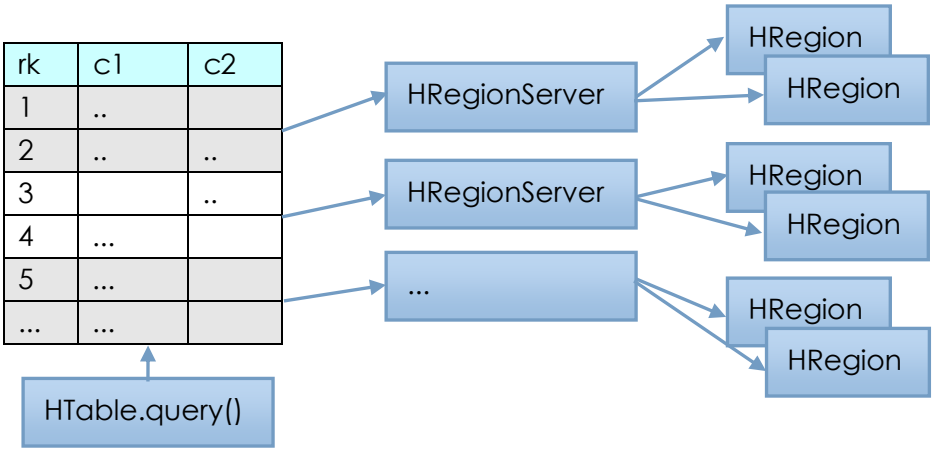
Step3: Socket 传输机制

Connection 主要负责传输. 主要成员 socket(java.net.Socket)是一条到 HBase Server 连接. 在这条连接上, Connection 是传输调用并接收结果.它也支持并发.前面提到的 calls 表存放了所有正在进行传输的 Call.每个 Call 都是从一特定的调用者线程中发起.但数据接收却不在调用者线程.

Connection 本身是 java.lang.Thread 的子类,在它本身的线程会在 socket 连接建立时启动,功能是循环检查当前有没有调用 Call 存在,如果有就试着从 socket 上读取结果.并分析出结果是对应到哪个(查 calls 表)Call,将结果放入 Call,并设置 Call 的完成标志,通知(call.notify())调用者线程.

Question5: 客户端的 RPC 容错和重试机制

从用户端看到 API 并不是直接的 HRegionInterface 或 HMasterInterface,而是 HTable 类已经包装的高级 API. HTable 这层的功能封装是较复杂的,因为这一层的操作会分配到不同 HBase 服务器上,比如说,从上层看只是对一个表的查询,但在 HTable 这层被分解成为了对不同 HRegionInterface(HRegionServer)的调用.因为一个表是有多个 Region 的,而不同的 Region 被分配到了不同的 Region Server 上了,而一个 HRegionInterface 又代表了一个 Region Server.另外,为了获得 Region 在服务器上的分布,还要扫描 Root 表和 Meta 表等等.



本文的重点在是在 HTable 的业务逻辑与 RPC 底层机制之间部分—容错机制.

HTable 所有引起 RPC 调用的方法,一般都会调用到 HConnectionManager.TableServers 的 getRegionServerWithRetries()这个方法. 以简单的 HTable.delete(Delete)这个函数为例,你可以看到一个对 HRegionInterface.delete 的调用被封装成了一个 org.apache.hadoop.hbase.client.ServerCallable 对象. 这个对象然后被作为参数传入了 getRegionServerWithRetires 方法调用.
下面的 HBase 代码版本为 hbase-0.94.12. HConnectionImplementation→TableServers
ServerCallable(HConnection).withRetries()→getRegionServerWithRetires(ServerCallable)

```
public void delete(final Delete delete) {
  new ServerCallable<Boolean>(connection, tableName, delete.getRow(), operationTimeout) {
        public Boolean call() throws IOException {
          server.delete(location.getRegionInfo().getRegionName(), delete);  →  HRegionInterface
          return null;
        }
      }.withRetries();
}
```

在这个方法内部,这个 ServerCallable 对象被调用到,成功了就返回,否则尝试调用多次,默认 10 次.
每次调用时的间隔还逐渐拉长.其中还将出错抛出的异常记到一个列表中,以便最终失败的时候分析原因.

考虑以下情况,当一个 Region 正被迁移,最初时 Delete 中的 row key 对象可能由 Region Server 1 来管理,但当调用发生后客户端收到了出错,因为此时这个 row key 从属的 Region 由 Region Server 2 来服务了.在这种重试机制下,都会经过两个流程

1) 调用 ServerCallable.instantiateServer(). 这步会重置这个 row key 所属的 location 及对应的 Region server.这样就可以获得 region server 2 了.
2) 调用 ServerCallable.call().重新发起调用.

这个机制也兼顾了一些特别情况,比如客户端可以收到一条称为 DoNotRetryIOException 的异常,这样客户端就不会再试了.举例来说,当执行 Scan 操作时,当 Region 迁移时,服务端会要求客户端在其它 Region server 上重启 Scan,而不要继续.

HBase 源码分析 3-从 Put 到 HFile

Hbase 插入数据的过程大致是：

1. 客户端提交请求给 region server（这中间会有作一些缓存）
2. region server 接收到请求，判断其是 put 请求，将其 put 到 memstore
3. 每次 memstore 的操作，都会检查 memstore 是否操作一个阈值，如果超过，就开始执行 flush()，这个 flush 其实就是将内存中的 KeyValue 对持久化到 HStore（也就是 HFile）上面

好了，下面看一下一条数据是怎么从 client 端到达 server 端，并且最终转换成 HFile 的吧~

Client 端(HTable)

```
1 public void put(final Put put) {
    doPut(put);  2
    if (autoFlush) {
      flushCommits();
    }
  }
  private void doPut(Put put) {
    validatePut(put);  3
    writeBuffer.add(put);  4
    currentWriteBufferSize += put.heapSize();
    if (currentWriteBufferSize > writeBufferSize) {
      flushCommits();
    }
  }
  public void flushCommits() throws IOException {
      Object[] results = new Object[writeBuffer.size()];
      this.connection.processBatch(writeBuffer, tableName, pool, results);  5
  }
```

1. 执行 Put 方法
2. Put 方法进一步调用 doPut(put)方法
3. 在 doPut 里面，会验证一个 put 的合理性（比如是否指定了 column）；然后会检查 KeyValue 的大小是否越界，这个可以通过配置参数 hbase.client.keyvalue.maxsize 来配置，默认是无限大的~
4. doPut 调用 writeBuffer.add(put)，将这个 put 写入到本地缓存，只有在以下几种情况下这个缓存才会 flush
   - 超过了 writeBufferSize，这个默认是从配置里面加载的，如果没有配置，就为 2097152；也可以通过调用 HTable 的方法 setWriteBufferSize()来改变
   - 设置了 autoflush，默认 autoflush 是打开的
   - 手动调用 flushCommits()方法
   - 调用 close()方法
5. 在 flush 方法中，通过执行 connection.processBatchOfPuts()，连接远程 region server，提交请求：

注：cilent 在执行插入的时候，会对最近使用的 region server 做 cache，如果有在 cache 有保存着相应的 region server 信息，就直接取出这个 region 信息，连接这个 region server。否则才对 master 执行一次 RPC，获得 region server 信息

客户端的操作，put，delete，get 都是封装在一个对象 Action 里面的每次提交，都是一系列的 Action 一起提交，也就是 MultiAction。

HConnection → processBatchCallback() → createCallable() → server.multi(MultiAction)

Server 端(HRegionServer+HRegion)

1. 执行 HRegionServer.multi(MultiActionmulti)，处理插入请求。
   - 取出每一个 action 对象，判断其属于哪种实例（ instanceof Delete/Put/Get），来执行特定的操作
   - 给每一个 put 分配一个 lock
   - 执行 HRegion.put()，正式进行 put。
2. 在 HRegion.put()方法中，又会一次调用以下方法
   - checkReadOnly()　　//检查 region 是否只读，如果只读，就会抛出异常
   - checkResources()
   - startRegionOperation()　　//获得锁
   - doMiniBatchPut(batchOp)
3. doMiniBatchPut(batchOp)方法会进行以下操作：
   - 获得锁
   - 写时间戳
   - 写 HLog
   - 写 Memstore：执行方法 applyFamilyMapToMemstore，这里面会调用 Store store = getStore(family); 获得 store 实例，然后调用 store 的 add 方法，添加每一个 kv 对到 store 里面。

注：在 HRegion 的 put 方法中，执行完 doMiniBatchPut 之后，会检查 memstore 的大小是否超过阈值，如果超过，就执行一次 flush，flush 的时候，会对 memstore 进行一个快照，就是暂时停止 memstore 服务，然后立即生成一个新的 memstore 对象，代替当前 memstore 接替当前 memstore 的工作，被替换的 memstore 会被写到 HStore（HFile）中。
Memstore 的 flush 是在方法 HRegion.internalFlushcache()里执行的。
切换 memstore 是非常快的，生成一个新对象，同时把 memstore 的指向直接切换到新的 kvset。

HBase 源码分析 4-Get 流程及 RPC 原理

客户端(HTable)

htable.get(Get)

```
public Result get(final Get get) throws IOException {
  return new ServerCallable<Result>(connection, tableName, get.getRow(), operationTimeout) {
        public Result call() throws IOException {
          return server.get(location.getRegionInfo().getRegionName(), get);
        }
      }.withRetries();
}
```

调用 get 方法后，客户端进入睡眠，睡眠时间为 pause * HConstants.RETRY_BACKOFF[ntries];
pause= HBASE_CLIENT_PAUSE（1 秒）
RETRY_BACKOFF[] = { 1, 1, 1, 2, 2, 4, 4, 8, 16, 32 };
有结果则中断执行返回 rpc 结果，否则重试十次（默认 DEFAULT_HBASE_CLIENT_RETRIES_NUMBER=10）
通过 HConnectionManager 的 getHRegionConnection 方法获取连接
通过 HRegionServer 的 get 方法获取结果

服务器(HRegionServer+HRegion)

当 regionserver 收到来自客户端的 Get 请求时，调用接口

```
public Result get(byte[] regionName, Get get) throws IOException {
    HRegion region = getRegion(regionName);
    return region.get(get, getLockFromId(get.getLockId()));
}
```

在 HRegion 中

```
private List<KeyValue> get(Get get, boolean withCoprocessor){
  List<KeyValue> results = new ArrayList<KeyValue>();
  Scan scan = new Scan(get);
  RegionScanner scanner = null;
    scanner = getScanner(scan);
    scanner.next(results, SchemaMetrics.METRIC_GETSIZE);
  return results;
}
```

Scan

Scan scan = new Scan(get);

```
public Scan(Get get) {
  this.startRow = get.getRow();
  this.stopRow = get.getRow();
  this.filter = get.getFilter();
  this.cacheBlocks = get.getCacheBlocks();
  this.maxVersions = get.getMaxVersions();
  this.tr = get.getTimeRange();
  this.familyMap = get.getFamilyMap();
}
```

会先根据设置的 columnFamily 存放 familyMap 对 ---- columnFamily:null

```
public Scan addFamily(byte [] family) {
  familyMap.remove(family);
  familyMap.put(family, null);
  return this;
}
```

如果查询的 family 不在 htableDescriptor 中，返回错误

RegionScanner(Impl)

```
public RegionScanner getScanner(Scan scan) throws IOException {
 return getScanner(scan, null);
}
protected RegionScanner getScanner(Scan scan, List<KeyValueScanner> additionalScanners) {
  return instantiateRegionScanner(scan, additionalScanners);
}
protected RegionScanner instantiateRegionScanner(Scan scan,List<KeyValueScanner> additionalScanners){
  return new RegionScannerImpl(scan, additionalScanners, this);
}
```

additionalScanners 为 null 所以在 RegionScannerImpl 的构造中只会使用 StoreScanner

RegionScannerImpl 是 HRegion 中的子类

```
RegionScannerImpl(Scan scan, List<KeyValueScanner> additionalScanners, HRegion region){
    List<KeyValueScanner> scanners = new ArrayList<KeyValueScanner>();
    if (additionalScanners != null) {
      scanners.addAll(additionalScanners);
    }
    for (Map.Entry<byte[], NavigableSet<byte[]>> entry : scan.getFamilyMap().entrySet()) {
      Store store = stores.get(entry.getKey());
      KeyValueScanner scanner = store.getScanner(scan, entry.getValue());
      scanners.add(scanner);
    }
  }
```

按照 familyMap 的数量存放对应数量的 StoreScanner

Store

Hregion initialize 时会对应每个 columnFamily 存放一个 stores

```
private long initializeRegionInternals(final CancelableProgressable reporter,){
      for (int i = 0; i < htableDescriptor.getFamilies().size(); i++) {
        Future<Store> future = completionService.take();
        Store store = future.get();
        this.stores.put(store.getColumnFamilyName().getBytes(), store);
      }
}
```

scanners 添加从 Store 中获取的 scanner

```
public KeyValueScanner getScanner(Scan scan,final NavigableSet<byte []> targetCols){
    return new StoreScanner(this, getScanInfo(), scan, targetCols);
}
```

Store.getScanner() → StoreScanner → 构造函数 getScannersNoCompaction → Store.getScanners

```
protected List<KeyValueScanner> getScanners(boolean cacheBlocks,
    boolean isGet, boolean isCompaction, ScanQueryMatcher matcher) {
  List<StoreFile> storeFiles = this.getStorefiles();
  List<KeyValueScanner> memStoreScanners = this.memstore.getScanners();

  // First the store file scanners
  List<StoreFileScanner> sfScanners = StoreFileScanner
    .getScannersForStoreFiles(storeFiles, cacheBlocks, isGet, isCompaction, matcher);
  List<KeyValueScanner> scanners = new ArrayList<KeyValueScanner>(sfScanners.size()+1);
  scanners.addAll(sfScanners);
  // Then the memstore scanners
  scanners.addAll(memStoreScanners);
  return scanners;
}
```

memStoreScanners 的获取:

```
List<KeyValueScanner> getScanners() {
    return Collections.<KeyValueScanner>singletonList(new MemStoreScanner());
}
```

Store 中为 StoreScanner 添加了 StoreFileScanner 和 memStoreScanner

scan

```
        scanner.next(results, SchemaMetrics.METRIC_GETSIZE);
```

现在分析 RegionScannerImpl 中的 next 方法，此时正式进入获取数据流程

```
    public boolean next(List<KeyValue> outResults, String metric)throws IOException {
      return next(outResults, batch, metric);       // apply the batching limit by default . batch默认为-1
    }
    public synchronized boolean next(List<KeyValue> outResults, int limit, String metric) {
      startRegionOperation();              //  为操作加读锁，lock.readLock().lock();
      return nextRaw(outResults, limit, metric);
    }
    public boolean nextRaw(List<KeyValue> outResults, int limit, String metric)    {
      boolean returnResult;
      if (outResults.isEmpty()) {
        // Usually outResults is empty. This is true when next is called to handle scan or get operation.
        returnResult = nextInternal(outResults, limit, metric);
      } else {
        List<KeyValue> tmpList = new ArrayList<KeyValue>();
        returnResult = nextInternal(tmpList, limit, metric);
        outResults.addAll(tmpList);
      }
      return returnResult;
    }
```

RegionScannerImpl.nextInternal → storeHeap.peek -- nextRow → storeHeap.next

```
    protected boolean nextRow(byte [] currentRow, int offset, short length) throws IOException {
      KeyValue next;
      while((next = this.storeHeap.peek()) != null && next.matchingRow(currentRow, offset, length)) {
        this.storeHeap.next(MOCKED_LIST);
      }
      return true;
    }
```

KeyValueHeap.next

```
  public boolean next(List<KeyValue> result, int limit, String metric) throws IOException {
3   InternalScanner currentAsInternal = (InternalScanner)this.current;
    boolean mayContainMoreRows = currentAsInternal.next(result, limit, metric);
2   KeyValue pee = this.current.peek();     → current就是前面getScanners放入的StoreFileScanner和MemStoreScanner
    this.heap.add(this.current);
    this.current = pollRealKV();
    return (this.current != null);
  }
```

1. 因 RegionScannerImpl 中 memStoreScanners 后添加，所以会先从 memStoreScanners 中查询，如果没有则从 StoreFileScanner 中查询 1
2. this.storeHeap 会不断 poll 出存储的 scanner(通过调用 peek).
3. RegionScannerImpl 的 storeHeap 为 KeyValueHeap，会强制转型 scanner 为 InternalScanner

总结下目前流程 get request -> regionServer -> region -> storeHeap -> scanner -> find row
但上述流程没有解释 reguest 是怎么找到 regionServer 去处理请求的，

RpcServer(线程)

下边我们在分析下服务器端服务在 HMaster 和 HRegionServer 启动时，中都会生成一个全局的 RpcServer

hmaster

hmaster 会使用 org.apache.hadoop.hbase.executor.ExecutorService 启动多种线程服务
HMaster.startServiceThreads():

MASTER_OPEN_REGION	（默认 5）
MASTER_CLOSE_REGION	（默认 5）
MASTER_SERVER_OPERATIONS	（默认 3）
MASTER_META_SERVER_OPERATIONS	（默认 5）
MASTER_TABLE_OPERATIONS	（单线程）
LogCleaner logCleaner	（单线程）
InfoServer infoServer	（master-status 等信息展示）
RpcServer rpcServer	（我们需要用的 rpc 服务）

RpcServer 是个接口，实现类为 HBaseServer << Server，启动时会开启 responder listener handlers 几种类去响应请求，如设置了 priorityHandlers 的数目，会另外启动 priorityHandlers，listener 监听端口，提供请求给 handlers，handlers 则调用 RpcEngine，反射出需要的方法并执行，通过 responder 写结果回去（this.responder.doRespond）。

HMaster 的 handlers 的个数由 hbase.master.handler.count
HRegionServer 的 handlers 的个数由 hbase.regionserver.handler.count 指定

HRegionServer

HRegionServer 的启动和 HMaster 类似，它启动以下线程：

RS_OPEN_REGION	（默认 3）
RS_OPEN_ROOT	（默认 1）
RS_OPEN_META	（默认 1）
RS_CLOSE_REGION	（默认 3）
RS_CLOSE_ROOT	（默认 1）
RS_CLOSE_META	（默认 1）
hlogRoller(daemon)	
cacheFlusher(daemon)	
compactionChecker(daemon)	
Leases	（它不是线程，会启动后台线程）
splitLogWorker	
rpcServer	

HBaseClient 和 HMaster 关系由 HMasterInterface 描述：

Clients interact with the HMasterInterface to gain access to meta-level HBase functionality, like finding an HRegionServer and creating/destroying tables.

HBaseClient 和 HRegionServer 关系由 HRegionInterface 描述：

Clients interact with HRegionServers using a handle to the HRegionInterface

参考资料：
http://zjushch.iteye.com/blog/1173304
http://www.spnguru.com/2010/07/hbase-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-rpc%E6%9C%BA%E5%88%B6-%E5%9F%BA%E7%A1%80/

HBase 源码分析 5-Master 启动过程

HMaster 是整个 hbase 中，负载较低的一个服务，他通过 zookeeper 负责整个集群 region 和 regionserver 的监控，负载均衡，以及 backup master，日志管理等功能。

构造函数

1. 创建 RPC 服务：

```
int numHandlers = conf.getInt("hbase.master.handler.count",conf.getInt("hbase.regionserver.handler.count", 25));
this.rpcServer = HBaseRPC.getServer(this, new Class<?>[]{HMasterInterface.class, HMasterRegionInterface.class},
    initialIsa.getHostName(), // This is bindAddress if set else it's hostname
    initialIsa.getPort(), numHandlers, 0, // we dont use high priority handlers in master
    false, conf, 0); // this is a DNC w/o high priority handlers
```

HBaseRPC.getServer → WritableRpcEngine.getServer → new Server → super → HBaseServer

a) 创建 Listener，主要负责为处理线程创建 job，该 Listener 会创建读取线程（Reader），线程个数为 ipc.server.read.threadpool.size，Reader 判断 Selector 中是否有数据，如果有，则把数据读出，然后加入到 call 队列或者优先队列中。

b) 创建 HbaseRpcMetric，负责统计数据收集

c) 创建 Responder，responder 负责将服务端的 reponser 数据异步写给客户端

d) 启动处理线程组，每一个线程组从 call 队列中读取数据，并调用对应的 call 方法（具体由客户端设定），处理完以后把每一个 call 的处理结果加入到 responder 会处理的 responseQueue 队列中

2. 创建 ZooKeeperWatcher，观察 hbase 节点，以及 hbase 下面的 master 节点，unassigned 节点，rs 节点，table 节点

3. 创建 MasterMetric 对象

```
this.zooKeeper = new ZooKeeperWatcher(conf, MASTER + ":" + isa.getPort(), this, true);
this.rpcServer.startThreads();
this.metrics = new MasterMetrics(getServerName().toString());
```

run

1. 基于 ZooKeeperWatcher 创建一个 ActiveMasterManager，加入到 zookeeper 的 listener 队列中，用于监视 zookeeper 节点状态

2. 如果是备份节点

a) 通过 zookeeper 监控节点状态，去写 master 的地址到 zookeeper 节点：

b) 如果写失败，则说明当前有别的 master，则一直等，直到 zookeeper 通知 master 节点状态发生变化；

c) 如果写成功，则备份节点成为主节点

3. 如果不是备份节点，如果自己作为 master 不 active，则循环等待，直到自己 active 为止，如果自己 active，那么进入 2.b.i

4. 在成为主 master 之前，初始化 master 的剩余几个类，以及启动一些服务，finishInitialization：

a) fileSystemManager：

i. 判断 HBASE_ROOT 有效性，等待 HDFS 退出 safemode

ii. 如果 HBASE_ROOT 不存在，则创建并写入 version，如果存在，则判断 version 是否和本 master 启动一致

iii. 如果 HBASE_ROOT 下不存在 root region，那么创建 root 和 meta 两个 region

iv. 创建 rootregion 是创建 regionId 为 0 的 region，这个 region 只负责 root 表的数据，他的列只有 info

v. 初始化 region，1)首先判断 region 对应的.Regioninfo 文件是否存在，如果大小非 0，那么就认为已经写过 regioninfo，否则写 regioninfo 数据，接着清除临时目录，2)接着对每一个 ColumnFamily 初始化 Store（Store 我们到介绍 region 的时候详细介绍，这里可以简单地理解为是一个映射表），3)根据初始化 Store 时读到的

maxSeqId，来进行日志回放，每一个 RegionServer 对应一个日志文件，这里我们也不进行深入展开，4)把上次遗留的 split 文件清理掉，以及 compaction 时的 merger 目录也删除掉

vi. 创建 meta region，第一个 meta region id 为 1，初始化过程和 5 一样

vii. 把 meta 信息加入到 root 表中

viii. 关闭 root region 和 meta region

b) Hconnection：

i. 从一个 LRU 的 conf hashcode=>TableServer 的 HashMap 中获取 Hconnection，如果 HashMap 中没有，则创建并加入到 map 中，创建过程如下：

ii. 从 conf 中读取一些客户端参数

iii. 设置 zooKeeperTrackers,包括一个 zookeeperWatcher，master 节点的地址监控 tracker，rootregion 的地址监控 tracker

c) ExecutorService:

i. 执行器服务，概括为一个线程池，一个 Event 队列，以及处理这个队列的处理器

d) ServerManger:

i. 管理 region server 的信息，HserverInfo,负载，以及死的 server。

ii. 他维护在线和离线的 server，处理 region server 的启动和关闭

e) CatalogServer:

i. 创建 root region 的地址监控 tracker，meta region 的地址监控 tracker，并启动他俩

f) AssignmentManager：

i. 用于管理 region 的分配

g) RegionServerTracker：

i. 启动通过 zookeerper 跟踪在线的 region server

h) ClusterStatusTracker:

i. 跟踪集群状态，并设置集群状态为 up，在这之前如果有 regionserver 启动，则 regionserver 会一直处于等待状态

i) 启动 ExecutorService 的不同服务，用于接收不同的请求类型（在上一节的 RpcServer(线程)提到）：

j) 等待 regionserver 汇报，ServerManager 统计

i. Server 数

ii. 每个 server 的 Region 数

k) 恢复日志，主要是针对还没有起来的 regionserver 的日志

i. 对于.logs 目录下，如果对应的文件名（其实就是主机名）没有对应的 regionserver，那么就需要将这部分的日志分摊(split log)到别的 region server 上

ii. Split log 简单地来说就是将属于一个 region 的日志重新从 regionserver 的日志中提取出来，写到一个新的文件中，以供加载 region 的时候使用，

iii. 具体地说，

1. 先创建一个 HlogSplitter，HlogSplitter 把 regionserver 的日志按记录读取出来，存放到一个 EntryBuffers 结构中，然后由一组写线程 writerThreads 读取这个结构，写入到 /HBASE_ROOT/TABLE_NAME/REGION_ENCODE_NAME/RECOVERED_EDITS_DIR/LOG_SEQ_NUM 文件中，如：hbase/some_table/2323432434/recovered.edits/2332

2. 在 split 的过程中，会遇到 EOFException、FileNotFoundException、IOException 等异常，EOFException 异常通常是由于文件截断造成的，比如 regionserver 异常 crash 了，FileNotFoundException 这种情况往往是由于 regionserver 在关闭之前把日志写完了，这时候这些日志实际上已经被持久化了，IOException 是剩余的情况，这种情况下，我们认为日志被损坏了
3. 把被处理的日志分别放到指定的 corrupt 目录和 oldLogDir 目录下，然后关闭 writerThreads 线程组

l) 再次确认 root 和第一个 meta 表已经分配：

i. 如果 root region 起来了，但是还在迁移，那么等待他迁移完成，此时 master 是被阻塞的
ii. 如果 meta region 起来了，但是还在迁移，那么等待他迁移完成，此时 master 也是被阻塞的

m) 如果一个 regionserver 都没有，AssignmentManager 重新分配所有的 region，否则处理 failover 状况

i. 重新分配所有 region 包括:
1. 清除目前的 region 分配状况
2. 重新分配所有非 disable 的表的 region，分配 region 有两种 balance 方法，一种是按照上次的分配方式，一种是 roundrobin 方式，可以通过 hbase.master.startup.retainassign 配置，如果配置成 true，则为上次的分配方式。分配方式由一个叫做 BulkAssigner 的对象负责。
ii. 处理 failover 状况
1. 首先获取未被分配的 region 对应的 server 列表，然后对每台 server 机器上的 region 进行处理，其实只是从 zookeeper 上面设置 region 的信息为 offline
2. 如果是因为分裂导致的 offlineregion，那么确认新 split 出来的 region 信息已经加入到 meta 表中
iii. 获取那些正在迁移的 region，将这些 region 信息更改加入到当前的 zookeerper 迁移列表中，交给 zookeeperWatcher 负责更新

n) 启动 balancer 线程，balancer 是一个后台定时启动的 chore

i. 如果 balancer 已经关闭，那么直接退出这次 balance
ii. 如果有 region 在迁移，也直接退出这次 balance
iii. 如果有死的节点，那么也直接退出这次 balance
iv. 获取目前的分配状况，然后为每一个分配生成一个 balance 计划，分别执行，balance 在讲 balancer 的时候再细讲

o) 启动一个 meta 表的 Janitor（看门人,catalogJanitorChore），然后扫描 meta 表，去收集那些未使用的 region，用于垃圾回收，细节这里也不再展开

p) 设置 master 状态为已经初始化

5. 然后就是循环 sleep，直到遇到 stop 命令
a) 停掉 balancerChore
b) 停掉 catalogJanitorChore
c) 等待所有 regionserver 停掉
d) 关掉 rpc 服务
e) 停止 logCleaner，infoServer，activeMasterManager，catalogTracker，serverManager，assignmentManager
f) 删除 HConnection 连接，关闭和 zookeeper 的连接

HBase 源码分析 6-RegionServer 上的 Get 全流程

当 regionserver 收到来自客户端的 Get 请求时，调用接口

```
public Result get(byte[] regionName, Get get){
   HRegion region = getRegion(regionName);
   return region.get(get, getLockFromId(get.getLockId()));
}
```

我们看 HRegion.get 接口，其首先会做 family 检测，保证 Get 中的 family 与 Table 的相符，然后通过 RegionScanner.next 来返回 result

而 Scanner 是 Hbase 读流程中的主要类，先做一个大概描述：
从 Scanner 的 scan 范围来分有 RegionScanner，StoreScanner，MemstoreScanner，HFileScanner；根据名称很好理解他们的作用，而他们之间的关系：RegionScanner 由一个或多个 StoreScanner 组成，StoreScanner 由 MemstoreScanner 和 HFileScanner 组成；

再看 RegionScanner 类的构造形成过程：

```
List<KeyValueScanner> scanners = new ArrayList<KeyValueScanner>();
for (Map.Entry<byte[], NavigableSet<byte[]>> entry :
      scan.getFamilyMap().entrySet()) {
      Store store = stores.get(entry.getKey());
      scanners.add(store.getScanner(scan, entry.getValue()));
}
this.storeHeap = new KeyValueHeap(scanners, comparator);
```

这段代码为 RegionScanner 类内部属性 storeHeap 初始化，其内容就是 Region 下面所有 StoreScanner 的和；storeHeap 是一个 KeyValueHeap，从字面可以理解 result 就是从中获取的

接着看 store.getScanner(scan, entry.getValue())即 StoreScanner 类的构造形成过程：

```
1   //StoreScanner is a scanner for both the memstore and the HStore files
2    List<KeyValueScanner> scanners = new LinkedList<KeyValueScanner>();
3        // First the store file scanners
4        if (memOnly == false) {
5           List<StoreFileScanner> sfScanners = StoreFileScanner
6           .getScannersForStoreFiles(store.getStorefiles(), cacheBlocks, isGet);
7
8           // include only those scan files which pass all filters
9           for (StoreFileScanner sfs : sfScanners) {
10             if (sfs.shouldSeek(scan, columns)) {
11                scanners.add(sfs);
12             }
13          }
```

```
14      }
15      // Then the memstore scanners
16      if ((filesOnly == false) && (this.store.memstore.shouldSeek(scan))) {
17            scanners.addAll(this.store.memstore.getScanners());
18      }
19  return scanners;
```

一般情况下 StoreScanner 中添加了 HFileScanner 和 MemStoreScanner；
StoreFileScanner 的内部属性包括 HFileScanner 和 Hfile.Reader，在添加前会根据 timestamp，columns，bloomfilter 过滤掉一部分

Scanner 构造完毕以后，当最上层的 RegionScanner.next 时，首先会先从 MemStoreScanner 中获取，如果没有或者版本数不足，则再从 HfileScanner 中获取，而从 HfileScanner 获取时，先查看是否在 blockcache 中，如果 MISS 则再从底层的 HDFS 中获取 block，并根据设置决定是否将 Block cache 到 LruBlockCache 中