# QTracer 源码分析



---


udpate：2016-02-28 22:04:17
contact：zhangxin.zhang@qunar.com


---

# 概述

首先什么是QTracer?

QTracer是TC组开发的分布式链路跟踪系统
这个无需多说,详见TC wiki http://wiki.corp.qunar.com/x/EAPFAw

分布式链路跟踪系统被广为熟知源于Google Dapper论文
http://research.google.com/pubs/pub36356.html

# 设计

QTracer是一个分布式链路跟踪系统,分布式服务的跟踪系统需要记录在一次特定的请求后系统中完成的所有工作的信息,将这些调用串成一条链路,从用户发起请求,经过分布式系统中各种服务调用

根据google的论文,一次完成的请求与响应构成一条完成的链路,中间各种服务的调用为子链路,每个链路抽象为一个span,拥有自己的spanid,所有的span通过唯一的标志qtracerid关联起来成为一条完整的调用链.

正如论文里提到的,qtracer是要对应用透明的,为了做到真正的应用级别的透明，所以核心跟踪代码肯定要做的很轻巧，然后把它植入到那些无所不在的公共组件种，比如线程调用、控制流以及RPC库。

qtracer也的确做到了足够轻巧,核心跟踪代码主要为两个方法

1. qunar.tc.qtracer.QTraceClient#startTrace()

开始一条链路

2. qunar.tc.qtracer.QTraceScope#close()
   结束一条链路,发送span数据到server

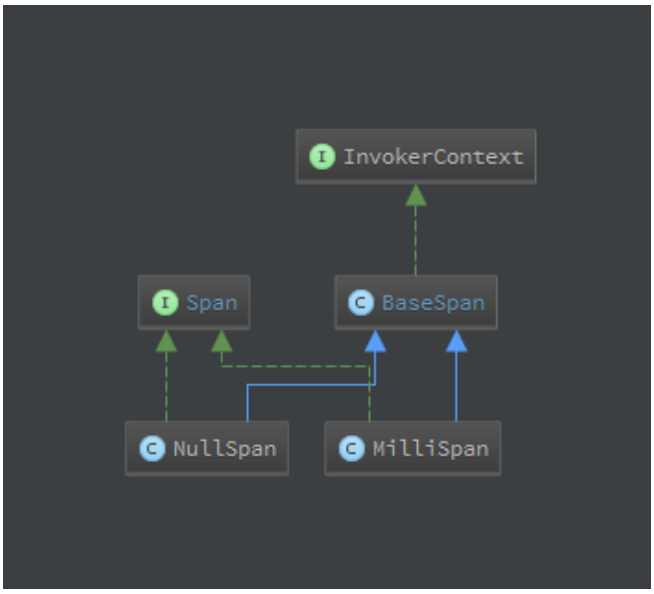
```
public interface QTraceClient {

    public QTraceScope startTrace(String desc);

    public QTraceScope startTrace(Span parent);

    public QTraceScope startTrace(String desc, TraceInfo info);

    public String getCurrentTraceId();

    public String getNextChildSpanId();

    public String getCurrentSpanId();
}
```

而且将代码植入到了公共组件,最基本的如埋点到qunar.dubbo,common-core中对http埋点,利用asm修改mysql.jdbc核心.class字节码文件埋点,还有common-concurrent中对线程池进行埋点,TC自己开发的redis包对redis埋点等等.

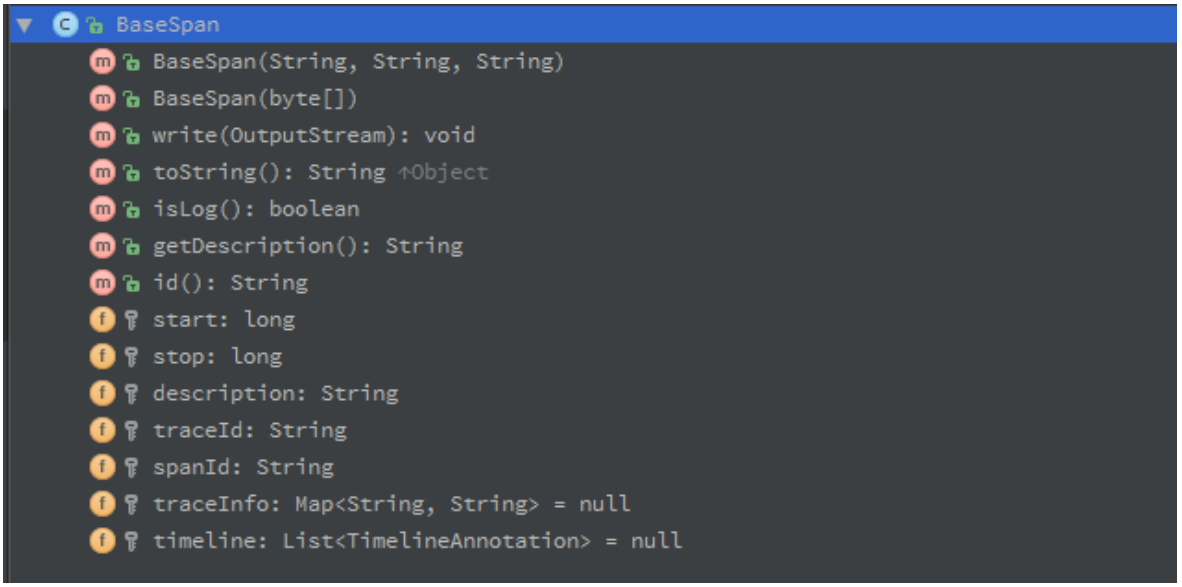
看看qtracer对span的抽象
主要定义一个span持有spanid和traceid

```
/**
 * User: zhaohuiyu
 * Date: 10/11/14
 * Time: 2:42 PM
 * len traceId spanId start stop desclen  desc   parentId kvsize [klen key vlen v
alue] tsize [time msglen msg]
 * 4    8       8     8     8     4    desclen     8       4     (klen+vlen)*kvsi
ze     4   (8+msglen)*tsize
 */
public interface Span {
    void stop();
    Span child(String description);
    void addKVAnnotation(String key, String value);
    void addTimelineAnnotation(String msg);
    String getChildNextId();
    /** appCode+ */
    String getTraceId();
    /** parentParentId.parentId.1 */
    String getSpanId();
}
```

看看实现




BaseSpan实现了span应该具有的基本实现,核心仍然是spanid和traceid,write()方法和构造函数实现了将一个span对象在DataOutputStream上的读/写,InvokerContext接口只有一个返回traceid的方法,提供给负载均衡策略使用,如一致性hash




MilliSpan实现需要采样时的span,NullSpan为不需要采样时的span

联系上文提到的两个核心方法

- `qunar.tc.qtracer.QTraceClient#startTrace()`
  开始一条链路就是生成一个span的过程,并且将该span通过ThreadLocal与当前线程关联起来,发生远程调用和线程切换都会为新线程生成一个新的子span,并且将traceid传递过去
- `qunar.tc.qtracer.QTraceScope#close()`
  结束一条链路就是将该线程的span从ThreadLocal中移除的过程,并且将span的数据通过netty发送给TC的服务端

那么现在做的就是在一次调用前后调用两个核心的方法,这就需要我们在公共组件中埋点了

# 字节码插桩

## Instrument代理

QTracer为了实现无入侵式的埋点,采用`java.lang.instrument`,用独立于应用程序之外的代理（agent）程序来监测和协助运行在JVM上的应用程序,这种监测和协助包括但不限于获取JVM运行时状态，替换和修改类定义等,qtracer基于这种代理然后使用`ASM`字节码框架来修改类定义,将两个核心方法添加进字节码文件.如修改`com.mysql.jdbc.ConnectionImpl`

> 什么是Instrumentation？
> http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html
> http://jiangbo.me/blog/2012/02/21/java-lang-instrument/

上文提到了两钟代理的方式,一种是在JVM启动时指定代理程序,第二种是在运行时制定,那么我们肯定是要用高大上的运行时制定咯.也就是agentmain方式.

为实现支持在 VM 启动之后启动代理，则必须满足以下条件：

1. 代理 JAR 的清单必须包含属性 Agent-Class。此属性的值是代理类 的名称。
2. 代理类必须实现公共静态 agentmain 方法。
3. 系统类加载器（ClassLoader.getSystemClassLoader）必须支持将代理 JAR 文件添加到系统类路径的机制。

关于第一点,在`qtracer-instrument-agent.jar`的pom文件中定义了Agent-Class

```
    <build>
        <finalName>qtracer-agent</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <archive>
                        <manifestEntries>
                            <Premain-Class>qunar.tc.qtracer.instrument.AgentMai
n</Premain-Class>
                            <Agent-Class>qunar.tc.qtracer.instrument.AgentMain</A
gent-Class>
                            <Can-Redefine-Classes>true</Can-Redefine-Classes>
                            <Can-Retransform-Classes>true</Can-Retransform-Classe
s>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

关于第二点,我们看看哪个类有agentmain方法:
qtracer代理类除了instrument定义的标准方法外,还提供了一个拿到Instrumentation实例的方法,用来在类外部实现修改.class文件的逻辑

```
public class AgentMain {
    private static Instrumentation inst;
    /** 命令行启动 */
    public static void premain(String agentArgs, Instrumentation instrumentation)
{
        inst = instrumentation;
    }

    /** 类加载调用 */
    public static void agentmain(String agentArgs, Instrumentation instrumentatio
n) {
        inst = instrumentation;
    }

    public static Instrumentation instrumentation() {
        return inst;
    }
}
```

qtracer拿到代理类的属性`Instrumentation`之后,添加类转换器,JVM每次加载一个.class文件,Instrumentation都会先调用类转化器的转换方法,修改特定类的字节码

```
        Instrumentation inst = instrumentation();//通过反射拿到Instrumentation实例
        inst.addTransformer(new QTraceClassFileTransformer(inst, configuratio
n));//添加转换器,修改字节码
```

代理jar和代理类找到了,那么如何在应用程序启动之后开启代理程序呢？ JDK6中提供了Java Tools API，可以满足这个需求。
https://blogs.oracle.com/CoreJavaTechTips/entry/the_attach_api

## Tools API

如参考文章提到的:
Attach API中的VirtualMachine代表一个运行中的VM。其提供了loadAgent()方法，可以在运行时动态加载`qtracer-instrument-agent.jar`。

common-core要求在web.xml中配置了一个必须在顶端的listener,这个listener监听了项目启动

`web.xml`中配置

```
<listener>
    <listener-class>qunar.ServletWatcher</listener-class>
</listener>
```

`在init()`方法中调用了这个方法

```
    private void startInstrument() {
        Instruments instruments = new Instruments();
        instruments.start();
    }
```

看看`start()`的实现

```
public void start() {
        if (!inited.compareAndSet(false, true)) return;

        Configuration configuration = new Configuration();//加载TC配置instrument.pr
operties
        if (!configuration.isInstrument()) {
            logger.warn("未开启instrument");
            return;
        }

        JavaAgentLoader loader = new JavaAgentLoader();
        boolean agentLoaded = loader.loadAgent();//找到当前运行的JVM,将代理jar附属上去
        if (!agentLoaded) {
            logger.warn("agent load failed");
            return;
        }
        Instrumentation inst = instrumentation();//通过反射拿到Instrumentation实例
        if (inst == null) {
            logger.warn("can not get instrumentation");
            return;
        }
        try {
            inst.addTransformer(new QTraceClassFileTransformer(inst, configuratio
n));//添加转换器,修改字节码
        } catch (Throwable ignore) {
            logger.error("add class transformer failed");
        }
    }
```

`loader.loadAgent()`中先找到当前正在运行的JVM,然后将代理jar附属上去

```
    void loadAgent() {
        VirtualMachine vm; //附属到当前运行JVM的VM

        if (AttachProvider.providers().isEmpty()) {
            String vmName = System.getProperty("java.vm.name");// Java 虚拟机实现名
称

            if (vmName.contains("HotSpot")) { //HotSpot JVM @link http://xiaomogu
i.iteye.com/blog/857821
                vm = getVirtualMachineImplementationFromEmbeddedOnes();
            } else {
                String helpMessage = getHelpMessageForNonHotSpotVM(vmName);
                throw new IllegalStateException(helpMessage);
            }
        } else {
            vm = attachToRunningVM();
        }

        loadAgentAndDetachFromRunningVM(vm);
    }
```

```
    private void loadAgentAndDetachFromRunningVM(VirtualMachine vm) {
        try {
            vm.loadAgent(jarFilePath, null); //把代理jar附到当前JVM上去
            vm.detach();
        } catch (AgentLoadException e) {
            throw new IllegalStateException(e);
        } catch (AgentInitializationException e) {
            throw new IllegalStateException(e);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

`instrumentation()`方法中通过反射拿到我们上文提到的代理类的`Instrumentation`属性,性,qtracer通过`java.lang.instrument.Instrumentation#addTransformer(java.lang.instrument.ClassFileTransformer)`方法添加字节码转化器

## ClassFileTransformer&ASM

上文提到为Instrumentation添加了ClassFileTransformer,这样JVM每次加载.class的时候都会被Instrumentation监测到,然后先经过ClassFileTransformer修改字节码,具体修改字节码的实现交由字节码框架ASM

> **什么是ASM?**
> ASM是一个java的字节码框架，它能被用来动态生成类或者增强既有类的功能。一般asm的应用场景主要在aop上，比如Spring在底层就是用了asm.
> http://wiki.corp.qunar.com/pages/viewpage.action?pageId=105919726

先看看`interface ClassFileTransformer`
jdk api中提到,一个提供此接口的实现以转换类文件的代理。转换在 JVM 定义类之前发生。

**参数：**

- loader - 定义要转换的类加载器；如果是引导加载器，则为 null
- className - 完全限定类内部形式的类名称和 The Java Virtual Machine
- Specification 中定义的接口名称。例如，“java/util/List”
- classBeingRedefined - 如果是被重定义或重转换触发，则为重定义或重转换的类；如果是类加载，则为 null
- protectionDomain - 要定义或重定义的类的保护域
- classfileBuffer - 类文件格式的输入字节缓冲区（不得修改）

**返回：**

- 一个格式良好的类文件缓冲区（转换的结果），如果未执行转换,则返回 null。

```
public interface ClassFileTransformer {
    byte[] transform(ClassLoader         loader,
                     String              className,
                     Class<?>            classBeingRedefined,
                     ProtectionDomain    protectionDomain,
                     byte[]              classfileBuffer)
        throws IllegalClassFormatException;
}
```

qtracer中`qunar.tc.qtracer.instrument.transform.QTraceClassFileTransformer`为此接口的实现类,里面完成了对mysql和postgreSQL中几个核心类字节码的修改

- com.mysql.jdbc.ConnectionImpl
- com.mysql.jdbc.PreparedStatement
- com.mysql.jdbc.StatementImpl
- org.postgresql.jdbc2.AbstractJdbc2Statement

```
qunar.tc.qtracer.instrument.transform.QTraceClassFileTransformer#transform

    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?> classB
eingRedefined, ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException {
        try {
            if (classBeingRedefined != null) return null; //不重新转换
            TraceClass traceClass = configuration.match(inst, this, className);
            if (traceClass == null) return classfileBuffer;//不是需要转换的文件直接返
回

            ClassReader traceClassReader = new ClassReader(classfileBuffer);//读字
节码抽象类
            ClassWriter traceClassWriter = new ClassWriter(ClassWriter.COMPUTE_MA
XS);//写字节码抽象类
            TraceClassVisitor traceClassVisitor = new TraceClassVisitor(new Check
ClassAdapter(traceClassWriter), traceClass);
            traceClassReader.accept(traceClassVisitor, 0);//使TraceClassVisitor修
改ClassReader读取的.class文件

            return traceClassWriter.toByteArray();
        } catch (Throwable e) {
            System.err.print("Class: ");
            System.err.print(className);
            System.err.print(", ClassLoader: ");
            System.err.print(loader);
            System.err.print(" transform failed.\n");
            e.printStackTrace(System.err);
            return classfileBuffer;
        }
    }
```

在ASM中,修改字节码的逻辑交由`qunar.tc.qtracer.org.objectweb.asm.ClassVisitor`实现

在qtracer中
`qunar.tc.qtracer.instrument.transform.TraceClassVisitor`继承了该抽象类

最后真正修改字节码的方法
为`qunar.tc.qtracer.instrument.transform.TraceMethodVisitor#visitCode`,核心的思路就是在获取Connection和执行sql的前后加上两个核心的方法感兴趣的同学可以git qtracer的源码进去看一下

```
        //QTraceClient client = QTraceClientGetter.getClient();
        traceMethod.visitMethodInsn(INVOKESTATIC, "qunar/tc/qtracer/impl/QTraceCl
ientGetter", "getClient", "()Lqunar/tc/qtracer/QTraceClient;", false);
        //QTraceScope scope = client.startTrace(method.getSignature());
        traceMethod.visitLdcInsn(method.getDescription());
        traceMethod.visitFieldInsn(GETSTATIC, TRACEUTILS_INTERNALNAME, "NEW_NO_TR
ACE", "Lqunar/tc/qtracer/TraceInfo;");
        traceMethod.visitMethodInsn(INVOKEINTERFACE, "qunar/tc/qtracer/QTraceClie
nt", "startTrace", "(Ljava/lang/String;Lqunar/tc/qtracer/TraceInfo;)Lqunar/tc/qtr
acer/QTraceScope;", true);
        //store scope to local variable
        traceMethod.visitVarInsn(ASTORE, scopeVarIndex);

        //scope.addAnnotation(Constants.QTRACE_TYPE,method.getType);
        emitAddKVAnnotation(scopeVarIndex, Constants.QTRACE_TYPE, method.getTyp
e());

        traceMethod.visitMethodInsn(INVOKEINTERFACE, QTRACE_SCOPE_INTERNALNAME,
"close", "()V", true);
```

最简单的修改字节码的方式是将源类和修改后类的.java文件用ClassReader将两个类的字节码打印出来,对比差异,然后再通过ClassWriter来修改源类的字节码

# HTTP插桩

既然是http插桩,记录一条完整的http调用链,那么我们在http请求的最开始就要开始记录trace,我们直接就会想到将http请求到达servlet之前拦截的filter,http插桩依赖于common-core包配置在web.xml最上方的

```
    <filter>
        <filter-name>watcher</filter-name>
        <filter-class>qunar.ServletWatcher</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>watcher</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

具体的逻辑我们来看filter的主要方法`qunar.ServletWatcher#doFilter`,这个方法完成了一次http调用链的start和close

1. start
   如果是根span,则生成新的span和新的traceid,如果是子span,则从报文的header中拿出traceid和spanid,然后start一条http调用链
2. close
   将span数据发送给TC的服务端,并且从MDC和当前线程中移除当前span

```
    /**
     * Filter mapping to all request, and response to some special URI
     */
    @Override
    public void doFilter(ServletRequest _request, ServletResponse _response, Filt
erChain chain) throws IOException,ServletException {
            ...
            QTraceWrapper wrapper = null;
            try {
                wrapper = new QTraceWrapper(request, response);//开始一个新链路,根或
者子链路,封装start
                ...
                chain.doFilter(request, response);
                ...
            } finally {
                QTracer.addKVAnnotation(Constants.HTTP_ANNO_RESP_STAT, String.val
ueOf(response.status));//记录额外信息,这里是http返回状态
                if (wrapper != null) {
                    wrapper.recycle();//关闭当前链路,发送数据,封装close
                }
            }
     }
```

## qunar.tc.qtracer.QTraceClient#startTrace()

`startTrace()`开始一条新的调用链,生成新的span,要么是一个根span,要么就是由根span或者根span的子span生成的子span,分别返回QTraceScope,持有QTraceScope来管理trace的状态和数据的发送,并且持有QTraceScope引用来管理trace的生命周期,在后来要把这个QTraceScope close掉
QTraceScope有两个实现类.

1. RootTScope
   根scope,表示当前是一个根调用链,根scope的构造函数中会生成根span,这个时候生成的spanid为1,并生成新的traceid
2. DefaultTScope
   DefaultTScope表示一个中间的调用链,持有父span,并且用父span生成当前线程持有的子span,拥有新的子spanid,traceid沿用父span

在http链路的filter中,`startTrace()`方法调用被封装在`QTraceWrapper`的构造函数中,我们来看看这个构造函数大致调用链

- `qunar.tc.qtracer.servlet.QTraceWrapper#QTraceWrapper`
  拿到qtracer的核心类QTraceClientGetter.getClient();
  - `qunar.tc.qtracer.servlet.QTraceWrapper#init`
    从http报文的header中取出traceid和spanid
  - `qunar.tc.qtracer.servlet.QTraceWrapper#startTrace`
    调用client.startTrace(),记录ip,协议等额外信息,把traceid放入response的header
    - `qunar.tc.qtracer.impl.AbstractQTraceClient#startTrace(TraceInfo)`
      该方法开始一次新的trace,生成新的spanid,沿用或者生成新的traceid

看看具体实现,首先是作为入口的`QTraceWrapper`构造函数,得到核心类`QTraceClient`–link detail–

```
    public QTraceWrapper(ServletRequest servletRequest, ServletResponse response)
{
        try {
            this.appName = Util.getAppName();
            this.client = QTraceClientGetter.getClient(); //根据 dev beta prod 返回
不同的client
        } catch (Throwable e) {
            throw new RuntimeException("初始化QTracer失败", e);
        }
        init(servletRequest, response);
    }
```

```
    /** 从http报文的header中取出traceid和spanid,这里getParameter()干嘛的? */
    private void init(ServletRequest servletRequest, ServletResponse response) {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletReque
st;
        String traceId = httpServletRequest.getHeader(Constants.TRACE_ID);//qtrac
eid
        String spanId = httpServletRequest.getHeader(Constants.SPAN_ID);//qspanid
        String requestURI = httpServletRequest.getRequestURI();
        String mustTraceId = httpServletRequest.getParameter(Constants.MUST_TRAC
E);//mqtrace
        if (Objects.equal(mustTraceId, Constants.MUST_TRACE_ID)) {// equal("on")
            traceId = mustTraceId;
            spanId = Constants.ROOT_SPANID;//根spanid == 1
            traceOn = true;
        }
        startTrace(traceId, spanId, requestURI, servletRequest, response);
    }
```

```
    private void startTrace(String traceId, String spanId, String requestURI, Ser
vletRequest servletRequest, ServletResponse response) {
        qTraceScope = client.startTrace(requestURI, new TraceInfo(traceId, spanI
d));//start一次新的trace
        qTraceScope.addAnnotation(Constants.QTRACE_IP, RequestUtil.getRealIP((Htt
pServletRequest) servletRequest));
        qTraceScope.addAnnotation(Constants.QTRACE_TYPE, Constants.QTRACE_TYPE_HT
TP);//记录额外信息
        qTraceScope.addAnnotation(Constants.QTRACE_APPLICATION, appName);
        if (traceOn) {
            ((HttpServletResponse) response).addHeader(Constants.TRACE_ID, clien
t.getCurrentTraceId());//在response中加入traceid,传递到另一个span中去
        }
    }
```

最核心的方法是`qunar.tc.qtracer.impl.AbstractQTraceClient#startTrace()`
首先从当前线程中的ThreadLocal查看当前链路是根链路还是子链路,然后生成不同的QTraceScope–link detail–,如果是生成RootTScope,那么会先获得当前链路的采样率,在RootTScope的构造函数中会生成根span,并且生成新的traceid,如果当前链路不是根链路,那么会返回DefaultTScope,沿用父链路的traceid,生成子span

```
    protected QTraceScope startTrace(String desc, String traceId, String spanId)
{
        //从threadlocal里取，如果取到则说明是同进程的
        Span currentSpan = traceStatus.getCurrentSpan();
        if (currentSpan == null) {
            Sample sample = getSample(desc, traceId);//获得采样率
            return new RootTScope(desc, traceStatus, spanReceiver, sample, traceI
d, spanId);//根链路生成根scope
        }
        String currentTraceId = Strings.isNullOrEmpty(traceId) ? currentSpan.getT
raceId() : traceId;
        //生成子链路的scope DefaultTScope
        return TraceId.isSample(currentTraceId) ? new DefaultTScope(desc, traceSt
atus, spanReceiver) : new MockTScope(desc, traceStatus, spanReceiver);
    }
```

我们重点看一下RootTScope的构造函数

1. 生成traceid和spanid,根span(根span为spanid为1的span)
2. 将traceid放入MDC,我们知道logback输出到日志中的KV都放在MDC中,所以日志能打印出traceid
3. RootTScope持有QTraceStatus和Receiver
4. 将根span放入ThreadLocal(由QTraceStatus持有)

```
    public RootTScope(String msg, QTraceStatus qTraceStatus, Receiver<BaseSpan> s
panReceiver, Sample sample, String traceId, String spanId, Span span) {
        if (span != null) {
            this.root = span;
        } else {
            traceId = TraceId.traceId(sample, traceId);//生成tracerid
            spanId = TraceId.spanId(spanId);//生成ROOT_SPANID值为1
            this.root = sample == Sample.NO ? new NullSpan(msg, traceId, spanId)
: new MilliSpan(msg, traceId, spanId);//生成span
        }

        if (sample != Sample.NO) {
            MDC.put(Constants.TRACE_ID, root.getTraceId());//将qtraceid放入MDC
        }

        this.traceStatus = qTraceStatus;//持有trace状态QTraceStatus
        this.spanReceiver = spanReceiver;//持有Receiver引用
        traceStatus.remove();
        traceStatus.setCurrentSpan(root);//将生成的根span通过ThreadLocal和当前线程关联
起来
    }
```

到此,一次http链路的start结束

## qunar.tc.qtracer.QTraceScope#close()

`qunar.tc.qtracer.QTraceScope#close` 方法结束当前链路,并发送span数据到TC,业务线即可根据traceid到tc的平台上去查询该条链路

`doFilter()`方法中finally语句块完成了对`close()`方法的调用,调用被封装在`recycle()`方法中

```
qunar.tc.qtracer.servlet.QTraceWrapper#recycle

    @Override
    public void recycle() {
        qTraceScope.close();//结束
        MDC.remove(Constants.TRACE_ID);//从MDC中移除qtraceid
        MDC.remove(Constants.MDC_KEY);//移除QTRACER
    }
```

真正的close逻辑在`qunar.tc.qtracer.QTraceScope#close`中,它有两个不同的子类实现

1. RootTScope根scope
2. DefaultTScope子scope

先看看RootTScope的实现
整个方法主要完成两件事情,第一是将当前的span数据通过netty发送给TC;第二是从MDC中删除qtraceid,并且从QTraceStatus持有的ThreadLocal中删除当前span

```
    @Override
    public void close() {
        defaultClose(true);
    }

    private void defaultClose(boolean send) {
        Span currentSpan = traceStatus.getCurrentSpan();//从QTraceStatus中拿到当前s
pan
        if (root != currentSpan) {//根scope持有的必须是根span
            log.debug("root != currentSpan, threadlocal is not right");
        }
        if (root != null) {
            if (send) {
                root.stop();//记录stop时间
                spanReceiver.receive((BaseSpan) root);//通过netty将span数据发送到TC
            }
            MDC.remove(Constants.TRACE_ID);
            traceStatus.remove();//从QTraceStatus中移除span
        }
    }
```

QTraceScope的介绍中提到该类持有`QTraceStatus`和`Receiver<BaseSpan>`,QTraceStatus持有链路状态,Receiver完成数据的发送,让我们看看
`qunar.tc.qtracer.spanreceiver.AbstractNettyReceiver#receive`的具体实现

```
    @Override
    public void receive(T t) {//T=BaseSpan
        checkClients();//检查NettyClient连接
        if (!Filter.getInstance().hasPermit()) {//检查TC是否为该应用打开qtracer开关,TC wiki写到的告知TC更改permit.properties
            return;
        }

        NettyClient<T> client = selectAvailable(t);//通过traceid负载均衡,选到合适的netty节点
        if (client == null) {
            log.info("discarded! No available client!");
            return;
        }

        client.write(t);//发送span数据
    }
```

receive中通过负载均衡策略选择合适的netty节点,然后调用`client.write(t)`发送数据,NettyClient的实现在QTracer-common包中,我们来看看这个方法的具体实现,主要逻辑为将span放入一个阻塞队列BlockingQueue,然后通过netty批量的发送到服务端

```
    public void write(T t) {
        if (!queue.offer(t)) {//阻塞队列满记录日志并且丢失该span
            log.info(this.getClass().getSimpleName() + " queue is full, discarded! ");
        }
        this.channel.eventLoop().execute(FLUSH_TASK);
    }

    private Runnable FLUSH_TASK = new Runnable() {
        @Override
        public void run() {
            if (!channel.isWritable()) return;//如果阻塞直接return
            if (queue.size() > 0) {
                int size = Math.min(batchSize, queue.size());//batchSize一个批次处理量
                List<T> spans = new ArrayList<T>(size);
                size = queue.drainTo(spans, size);
                if (size > 0) channel.writeAndFlush(spans);//批量span数据发送到TC
            }
        }
    };
```

DefaultTScope的实现与RootScope的实现一样,只是DefaultTScope发送完span数据后将当前TheadLocal的span设置为父span,我们不再重复阐述DefaultTScope

到这里,一次http链路结束

# QTraceStatus

通过ThreadLocal持有当前的trace状态,即持有span,span中持有traceid和spanid

> 待更新…

# QTraceScope

持有当前span链路的生命周期,通过管理QTraceStatus和Receiver

> 待更新…

# 附录:

## Configuration字节码插桩文件配置

`qunar.tc.qtracer.instrument.Instruments#start`中生new出了管理配置文件的对象
`Configuration configuration = new Configuration();`
configuration中持有了对TC instrument.properties文件的qconfig引用，里面配置了需要进行字节码插桩的类
**Reference**