部署上线前准备

1. 索引

> *开发完成后，到了上线环节，需要检查mysql的索引是否设置合理，如果后续数据量多了在设置索引，那么用时会比较长，需要进行停机维护，造成不好的用户体验*

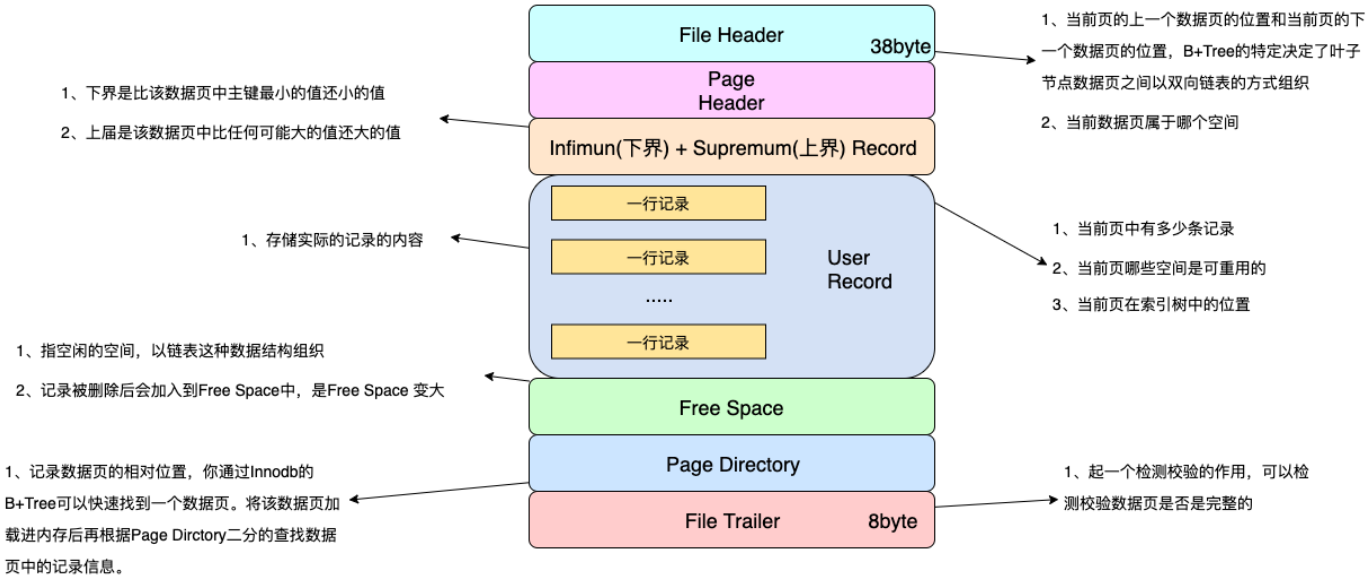
合理设置索引，可以有效的提高查询性能

索引的设计原则：

1. 每个表都必须有自增主键
    1. 有顺序，磁盘顺序读写 速度快
    2. 插入数据的时候，会插入到最后（B+Tree结构），减少了数据移动
    3. 减少页分裂（B+Tree结构）
2. 常做为查询条件的字段，排序，分组的字段建立索引
    1. 提高查询效率
3. 索引字段的选择尽量不使用 字段长度较长的
4. 数据量小的表 不建立索引
5. 限制索引的数量 不是越多越好 （通常建议在6个以内）
6. 写比多，并且写频繁的表不建议加索引
7. 不要在区分度低的字段建立索引
    1. 比如性别，只有 男 女 未知 三个值，索引完全起不到优化作用
8. 联合索引的创建要符合最左原则
    1. 遇到范围查询 索引失效
    2. 比如：a = 1 and b = 2 and c > 3 and d = 4，如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
    3. 比如: a = 1 and b > 3 and c < 4 , 我们可以对(a,b) 或者（a,c）建索引，都可以，如何选择，可以看 b和c谁的区分度高

1.1 mysql数据页

> *InnoDB从磁盘中读取数据的最小单位是数据页。而你想得到的id = xxx的数据，就是这个数据页众多行中的一行。*



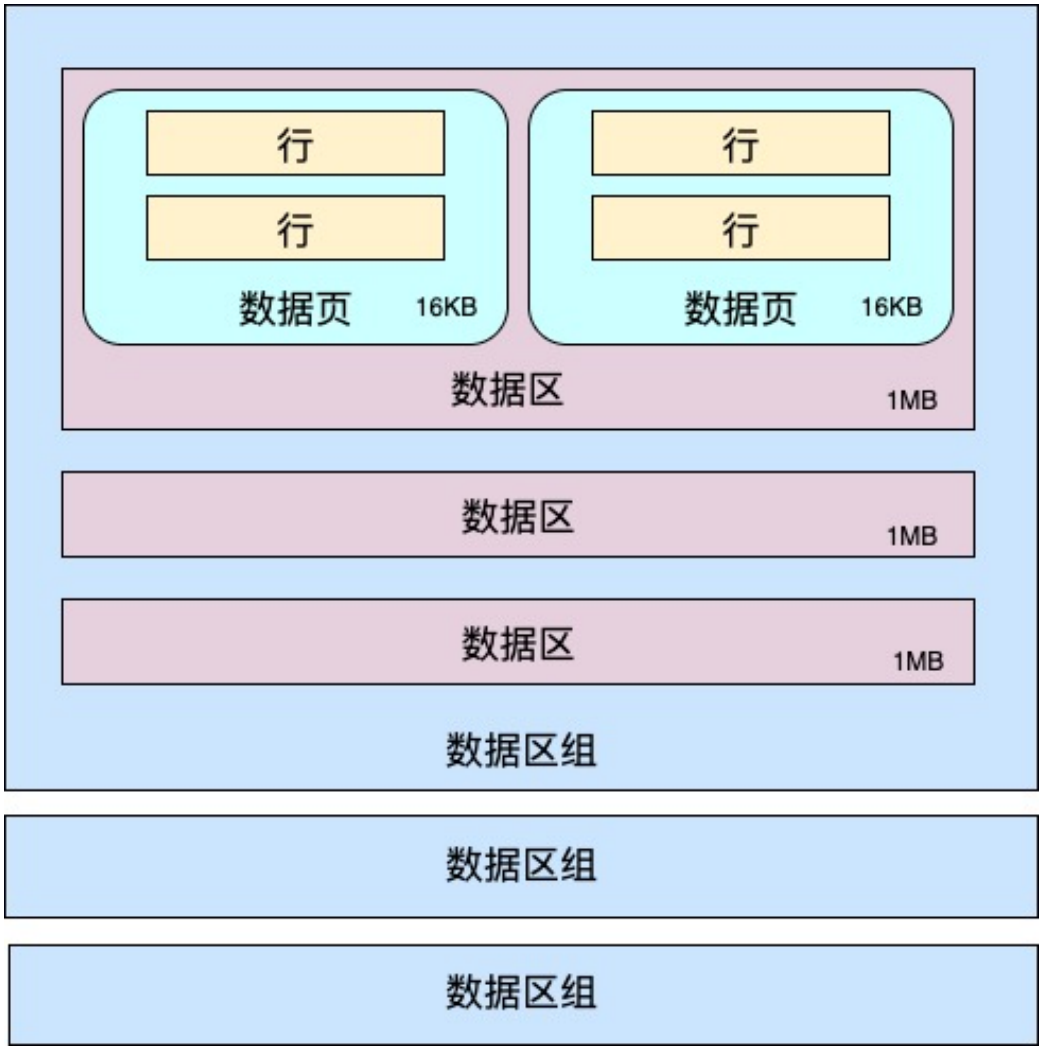
> *在InnoDB存储引擎中，数据页是InnoDB磁盘管理的最小的数据单位，数据页的默认大小为16KB。*

在MySQL5.6中：你可以通过参数`innodb_page_size`设置每个数据页的大小为4KB、8KB、16KB。一旦设置完成后，所有表中的数据页大小都将是你设置的值且不可变。不论你将`innodb_page_size`设置成多大，一个区（extent）1MB的事实都不会改变。

在MySQL5.7.6中：允许你将`innodb_page_size` 设置成 32KB、64KB大小。对于32KB大小的数据页来说区的大小被调整成2MB。对于64KB大小的数据页来说，区的大小被调整成4MB。

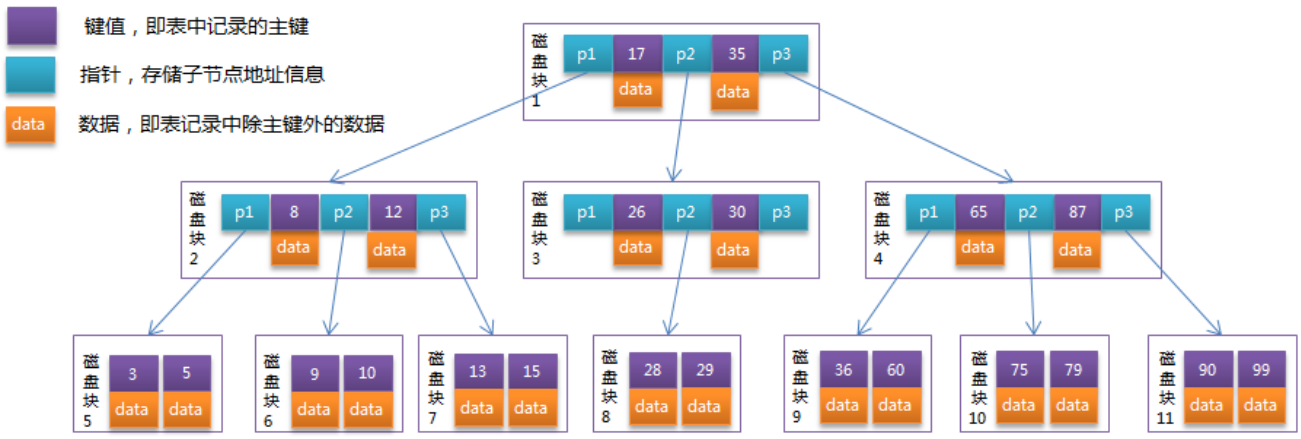
1.2 数据区

> *在MySQL的设定中，同一个表空间内的一组连续的数据页为一个extent（区），默认区的大小为1MB，页的大小为16KB。16*64=1024，也就是说一个区里面会有64个连续的数据页。连续的256个数据区为一组数据区。*



1.3 B-TREE和B+TREE

1. B-Tree(读B树,中间是连接符，不是减号)



> *以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35*

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

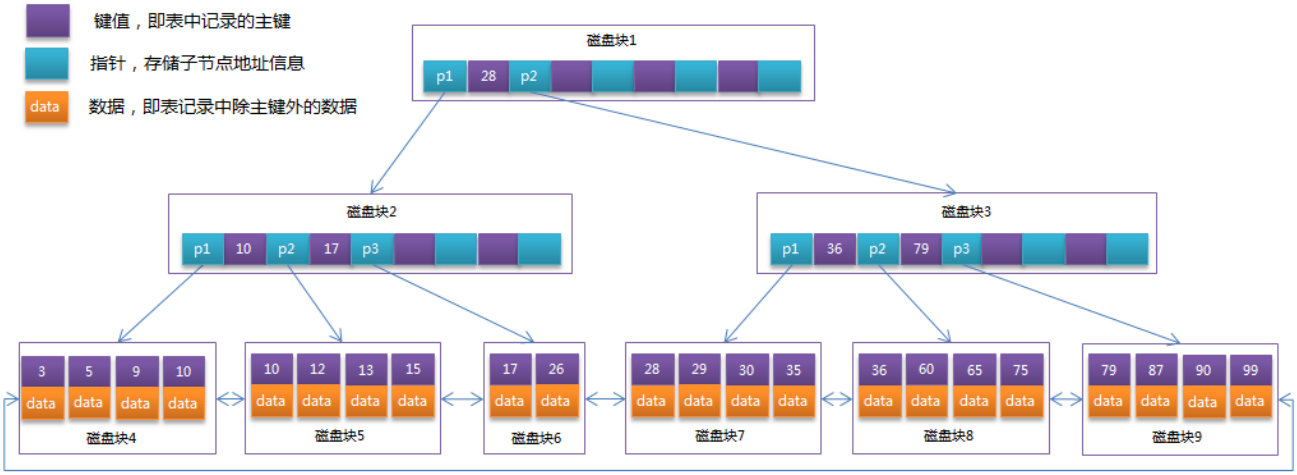
分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。

2. B+Tree

> *B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。*

B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。



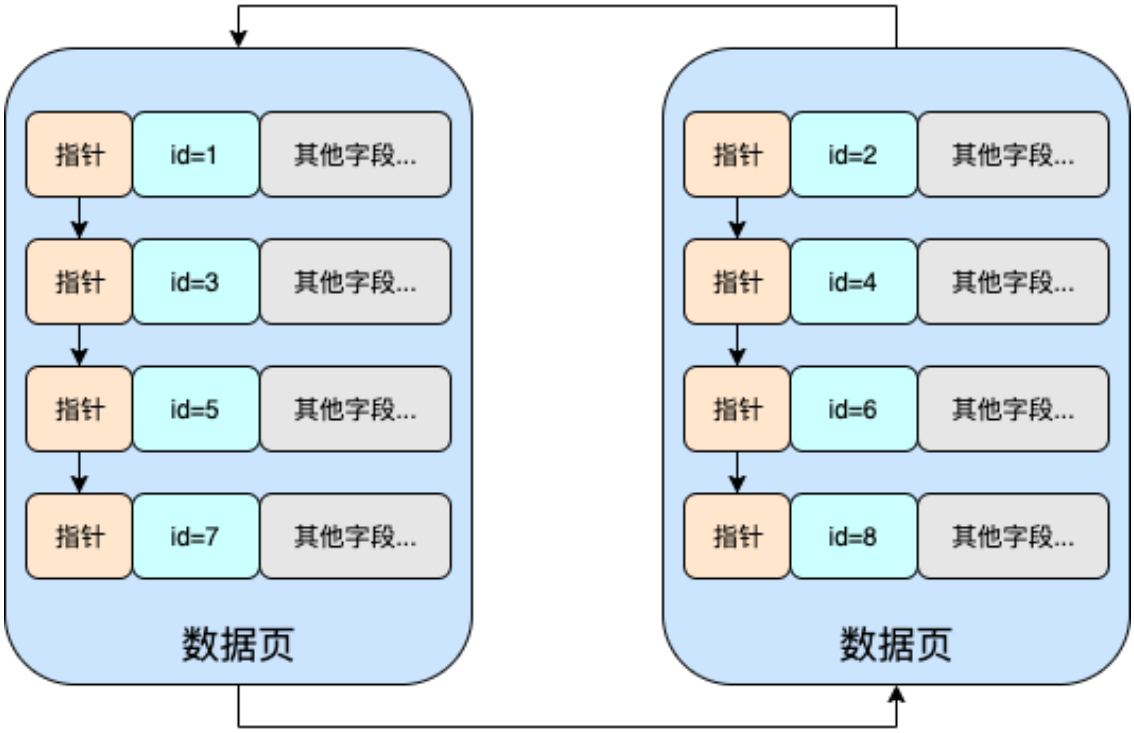
3. 由于非叶子节点不存储数据，所以可以存储大量的键，树的深度会减少，代表会进行较少的磁盘IO
4. 叶子节点使用链表，能够很好的支持范围查询
5. 充分利用空间局部性原理，适合磁盘存储
    1. 磁盘IO是一个比较耗时的操作，而操作系统在设计时则定义一个空间局部性原则，局部性原理是指CPU访问存储器时，**无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中**。
    2. 操作系统的文件系统中，数据也是按照page划分的，一般为4k或8k。**当计算机访问一个地址数据时，不仅会加载当前数据所在的数据页，还会将当前数据页相邻的数据页一同加载到内存**。而这个过程实际上只发生了1次磁盘IO，这个理论对于索引的数据结构设计非常有帮助。

1.4 数据页分裂问题

> *假设你现在已经有两个数据页了。并且你正在往第二个数据页中写数据。*

假设你自定义了主键索引，而且你自定义的这个主键索引并不一定是自增的

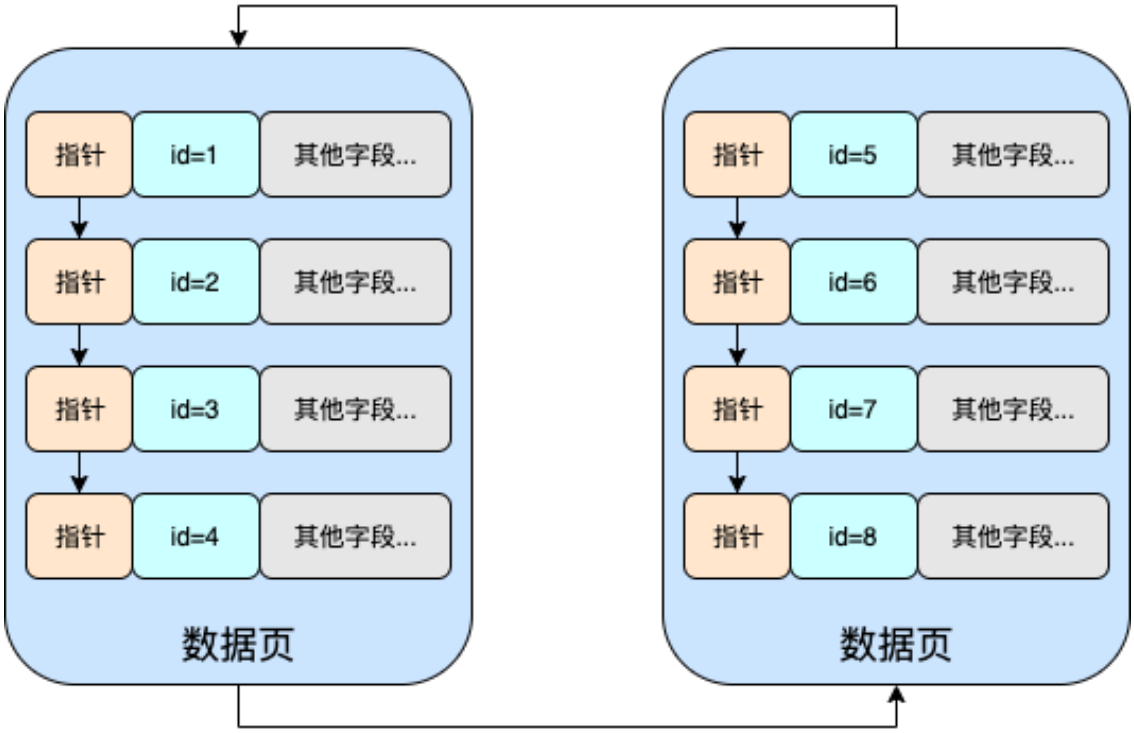
会出现如下情况：



> *随着你将数据写入。就导致后一个数据页中的所有行并不一定比前一个数据页中的行的id大。*

这时就会触发页分裂的逻辑

页分裂的目的就是保证：后一个数据页中的所有行主键值比前一个数据页中主键值大。



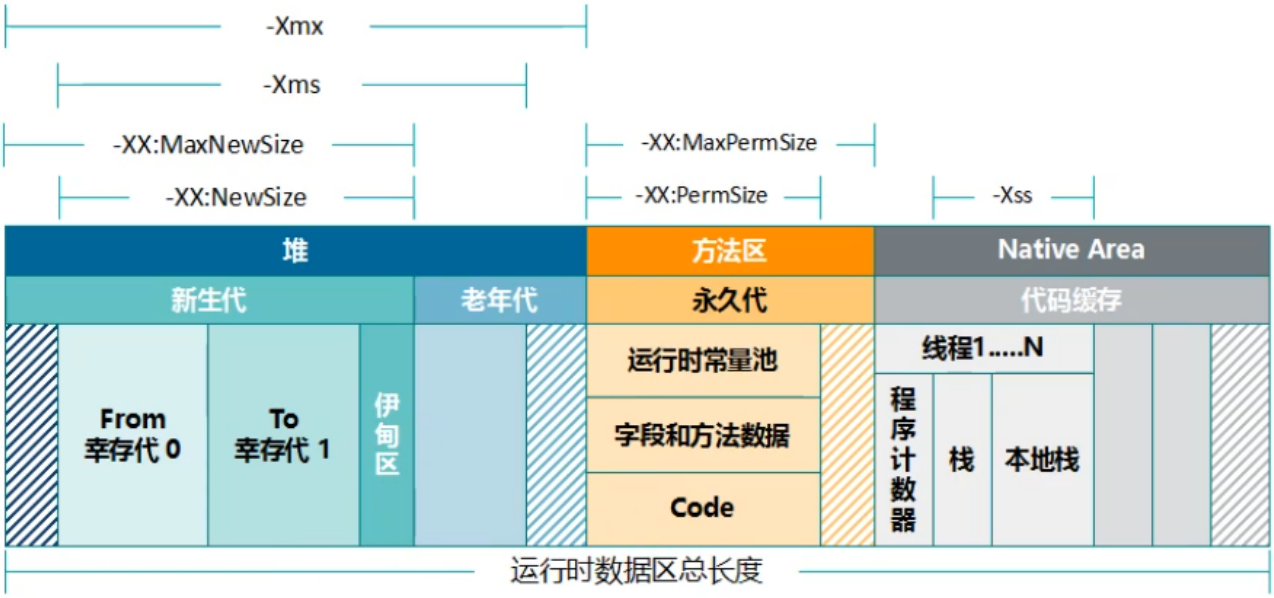
> *如果使用主键索引，就会减少页分裂*

2. JVM调优

> *上线之前，需要预估系统的访问量，设置合理的JVM参数*

2.1 分代回收

> *首先我们需要明白，JVM是在内存当中的，我们程序运行的过程当中，会持续的在内存中占用空间，有些对象使用完成之后，就不会在被使用了，那么应该被回收掉，释放内存空间，保证程序的运行。*



1. 年轻代（新生代）

> *新生代主要用来存放新生的对象。一般占据堆空间的1/3。在新生代中，保存着大量的刚刚创建的对象，但是大部分的对象都是朝生夕死，所以在新生代中会频繁的进行MinorGC，进行垃圾回收。新生代又细分为三个区：Eden区、SurvivorFrom、ServivorTo区，三个区的默认比例为：8：1：1。*

    1. **Eden区**

    > *Java新创建的对象绝大部分会分配在Eden区（如果对象太大，则直接分配到老年代）。当Eden区内存不够的时候，就会触发MinorGC（新生代采用的是复制算法），对新生代进行一次垃圾回收。*

    2. **SurvivorFrom区和To区**

    > *在GC开始的时候，对象只会存在于Eden区和名为From的Survivor区，To区是空的，一次MinorGc过后，Eden区和SurvivorFrom区存活的对象会移动到SurvivorTo区中，然后会清空Eden区和SurvivorFrom区，并对存活的对象的年龄+1，如果对象的年龄达到15，则直接分配到老年代。MinorGC完成后，SurvivorFrom区和SurvivorTo区的功能进行互换。下一次MinorGC时，会把SurvivorTo区和Eden区存活的对象放入SurvivorFrom区中，并计算对象存活的年龄。*

2. 老年代

> *老年代主要存放应用中生命周期长的内存对象。老年代比较稳定，**不会频繁的进行MajorGC**。而在**MaiorGC之前才会先进行一次MinorGC**，使得新生的对象进入老年代而导致空间不够才会触发。当无法找到足够大的连续空间分配给新创建的较大对象也会提前触发一次MajorGC进行垃圾回收腾出空间。*
>
> *在老年代中，MajorGC采用了**标记–清除算法**：首先扫描一次所有老年代里的对象，标记出存活的对象，然后回收没有标记的对象。**MajorGC的耗时比较长**。因为要扫描再回收。**MajorGC会产生内存碎片**，当老年代也没有内存分配给新来的对象的时候，就会抛出**OOM（Out of Memory）**异常。*

3. 永久代

> *永久代指的是永久保存区域。*
>
> *主要存放**Class和Meta（元数据）**的信息。*
>
> *Classic在被加载的时候被放入永久区域，它和存放的实例的区域不同，在Java8中，永久代已经被移除，取而代之的是一个称之为“元数据区”（元空间）的区域。*
>
> *元空间和永久代类似，都是对JVM中规范中方法的实现。不过元空间与永久代之间最大的区别在于：**元空间并不在虚拟机中，而是使用本地内存**。因此，默认情况下，元空间的大小仅受本地内存的限制。类的元数据放入native memory，字符串池和类的静态变量放入java堆中。这样可以加载多少类的元数据就不再由MaxPermSize控制，而由系统的实际可用空间来控制。*
>
> *采用元空间而不用永久代的原因：*
>
> - *为了解决永久代的OOM问题，元数据和class对象存放在永久代中，容易出现性能问题和内存溢出。*
> - *类及方法的信息等比较难确定其大小，因此对于永久代大小指定比较困难，大小容易出现永久代溢出，太大容易导致老年代溢出（堆内存不变，此消彼长）。*
> - *永久代会为GC带来不必要的复杂度，并且回收效率偏低。*

2.2 FullGC

MinorGC: 年轻代回收

Major GC：老年代回收

FullGC：年轻代，老年代，永久代都回收

触发FullGC的条件：

1. System.gc()

2. 老年代空间不足
3. 永久代空间不足
4. gc担保失败，进行MinorGC之前会检查老年代是否有足够的连续空间大于平均历次晋升到老年代大小，如果小于 则进行FullGC

调优的目的是尽量少的发生fullGC，FullGC会发生STW（世界停顿），影响系统性能

2.3 如何确定参数

> *JVM最优的参数，最好是在应用上线前就确定好，我们首先预估单机应用需要能承载的最大量级，然后进行压测，根据日志来进行调优*

2.3.1 项目部署到虚拟机

> *我们克隆一个虚拟机，将其中的部署软件都清空，部署springboot程序上去*

做一个新的配置文件，application-prod.properties，将其中的数据库，redis，等连接修改一下

1. 打包

   在api的模块依赖加入maven打包插件

```
<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <version>3.1.0</version>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

2. 如果选择连本机数据库，记得开启允许所有ip访问

```
update mysql.user set host = '%' where user ='root';
flush privilege;
```

3. 在虚拟机上创建目录，并将sso，web，sso-provider这些jar包上传
4. 创建启动脚本 并 chmod +x sso-api.sh

   sso-api.sh

```
#!/bin/sh
#这里可替换为你自己的执行程序，其他代码无需更改
APP_NAME=sso-api.jar
#使用说明，用来提示输入参数
usage() {
    echo "Usage: sh demo.sh [start|stop|restart|status]"
    exit 1
}

#检查程序是否在运行
is_exist() {
     pid=`ps -ef | grep $APP_NAME | grep -v grep | awk '{print $2}'
`
     echo "pid==${pid}"
     #如果不存在返回1，存在返回0
     if [ -z "${pid}" ]; then
        echo '不存在，没有启动,准备启动'
             return 1
        else
             return 0
        fi
}

#启动方法
start() {
    echo "*********check is_exist in first*****"
    is_exist
    if [ $? -eq "0" ]; then
       echo "${APP_NAME} is already running. pid=${pid} ."
       kill $pid
       sleep 5s
       kill -9 $pid
       echo "kill pid " $pid
    else
       echo "${APP_NAME} 开始启动.... ."
       nohup java -jar -Xmx512m -Xms512m -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=sso.dump  -
XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:sso-api.gc
$APP_NAME --spring.profiles.active=prod > sso.log 2>&1 &
```

```
        echo "启动完成"

        fi
}

#停止方法
  stop() {
       is_exist
       if [ $? -eq "0" ]; then
            kill -9 $pid
       else
            echo "${APP_NAME} is not running"
       fi
}

#输出运行状态
status() {
    is_exist
    if [ $? -eq "0" ]; then
       echo "${APP_NAME} is running. Pid is ${pid}"
    else
       echo "${APP_NAME} is not running."
    fi
}

#重启
restart() {
   stop
   start
}

#根据输入参数，选择执行对应方法，不输入则执行使用说明
case "$1" in
     "start")
     start
     ;;
     "stop")
     stop
     ;;
     "status")
     status
     ;;
     "restart")
     restart
     ;;
```

```
    *)
    usage
    ;;
esac
```

sso-provider.sh:

```
#!/bin/sh
#这里可替换为你自己的执行程序，其他代码无需更改
APP_NAME=sso-provider.jar
#使用说明，用来提示输入参数
usage() {
    echo "Usage: sh demo.sh [start|stop|restart|status]"
    exit 1
}

#检查程序是否在运行
is_exist() {
    pid=`ps -ef | grep $APP_NAME | grep -v grep | awk '{print $2}'
`
    echo "pid==${pid}"
    #如果不存在返回1，存在返回0
    if [ -z "${pid}" ]; then
       echo '不存在，没有启动,准备启动'
            return 1
       else
            return 0
       fi
}

#启动方法
start() {
    echo "*********check is_exist in first*****"
    is_exist
    if [ $? -eq "0" ]; then
       echo "${APP_NAME} is already running. pid=${pid} ."
       kill $pid
       sleep 5s
       kill -9 $pid
       echo "kill pid " $pid
    else
       echo "${APP_NAME} 开始启动 ."
```

```
        nohup java -jar -Xmx512m -Xms512m -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=sso-provider.dump
 -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:sso-provider.gc
$APP_NAME --spring.profiles.active=prod > sso-provider.log 2>&1 &
        echo "启动完成"

        fi
}

#停止方法
  stop() {
       is_exist
       if [ $? -eq "0" ]; then
            kill -9 $pid
       else
            echo "${APP_NAME} is not running"
       fi
}

#输出运行状态
status() {
    is_exist
    if [ $? -eq "0" ]; then
       echo "${APP_NAME} is running. Pid is ${pid}"
    else
       echo "${APP_NAME} is not running."
    fi
}

#重启
restart() {
   stop
   start
}

#根据输入参数，选择执行对应方法，不输入则执行使用说明
case "$1" in
     "start")
     start
     ;;
     "stop")
     stop
     ;;
     "status")
     status
```

```
    ;;
    "restart")
    restart
    ;;
    *)
    usage
    ;;
esac
```

web-api.sh:

```
#!/bin/sh
#这里可替换为你自己的执行程序，其他代码无需更改
APP_NAME=web-api.jar
#使用说明，用来提示输入参数
usage() {
    echo "Usage: sh demo.sh [start|stop|restart|status]"
    exit 1
}

#检查程序是否在运行
is_exist() {
    pid=`ps -ef | grep $APP_NAME | grep -v grep | awk '{print $2}'
`
    echo "pid==${pid}"
    #如果不存在返回1，存在返回0
    if [ -z "${pid}" ]; then
       echo '不存在，没有启动,准备启动'
            return 1
       else
            return 0
       fi
}

#启动方法
start() {
    echo "*********check is_exist in first*****"
    is_exist
    if [ $? -eq "0" ]; then
       echo "${APP_NAME} is already running. pid=${pid} ."
       kill $pid
       sleep 5s
       kill -9 $pid
```

```
        echo "kill pid " $pid
    else
        echo "${APP_NAME} 开始启动 ."
        nohup java -jar -Xmx512m -Xms512m -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=web-api.dump  -
XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:web-api.gc
$APP_NAME --spring.profiles.active=prod > web-api.log 2>&1 &
        echo "启动完成"

        fi
}

#停止方法
  stop() {
      is_exist
      if [ $? -eq "0" ]; then
           kill -9 $pid
      else
           echo "${APP_NAME} is not running"
      fi
}

#输出运行状态
status() {
    is_exist
    if [ $? -eq "0" ]; then
        echo "${APP_NAME} is running. Pid is ${pid}"
    else
        echo "${APP_NAME} is not running."
    fi
}

#重启
restart() {
   stop
   start
}

#根据输入参数，选择执行对应方法，不输入则执行使用说明
case "$1" in
     "start")
     start
     ;;
     "stop")
     stop
```

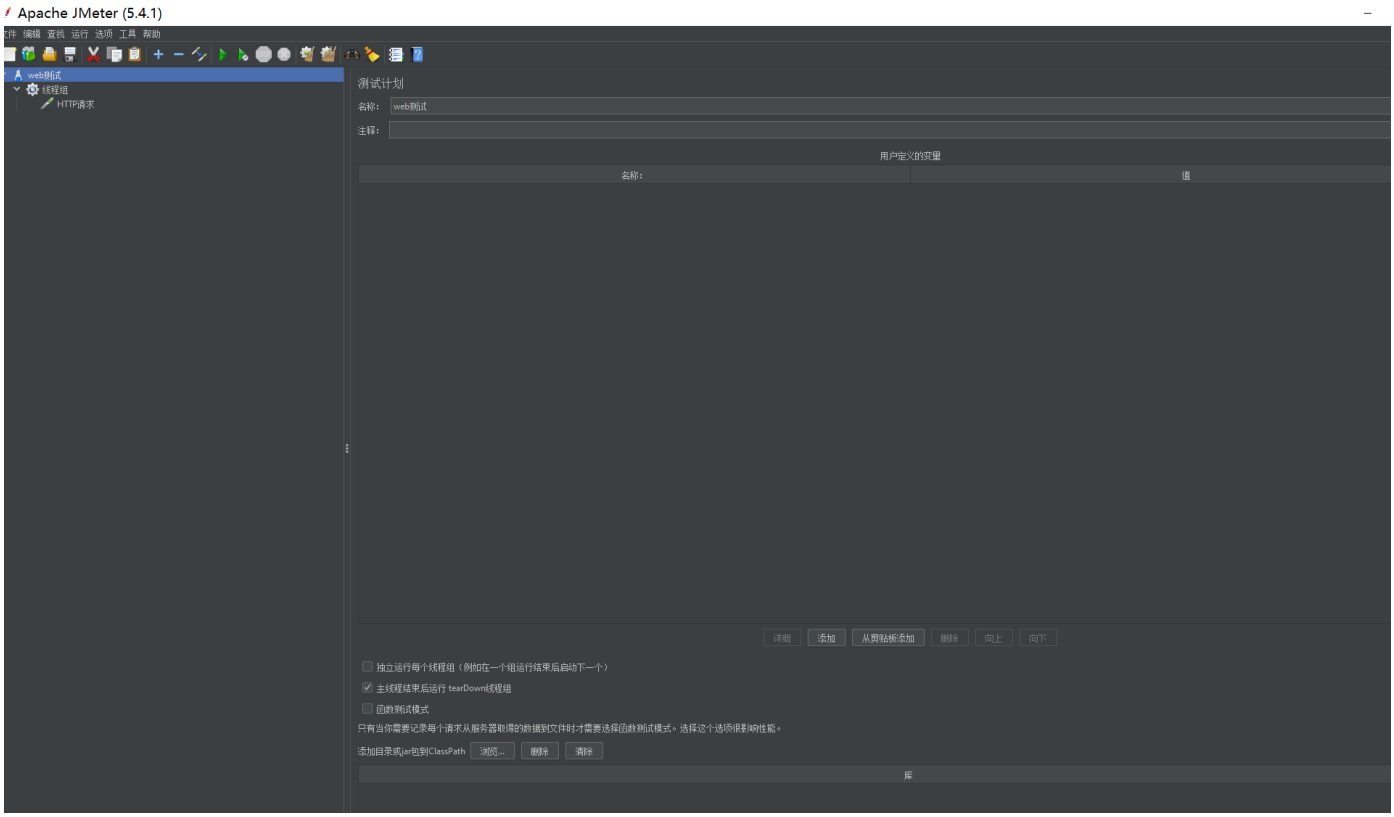
```
        ;;
        "status")
        status
        ;;
        "restart")
        restart
        ;;
        *)
        usage
        ;;
esac
```

2.3.2 压测

> *上方我们部署了程序，并将gc信息打印在了日志当中，接下来我们对web应用的查询课程列表接口进行压测，并且查看对应的gc日志*

资料中有提供jmeter，我们使用它来进行压测。

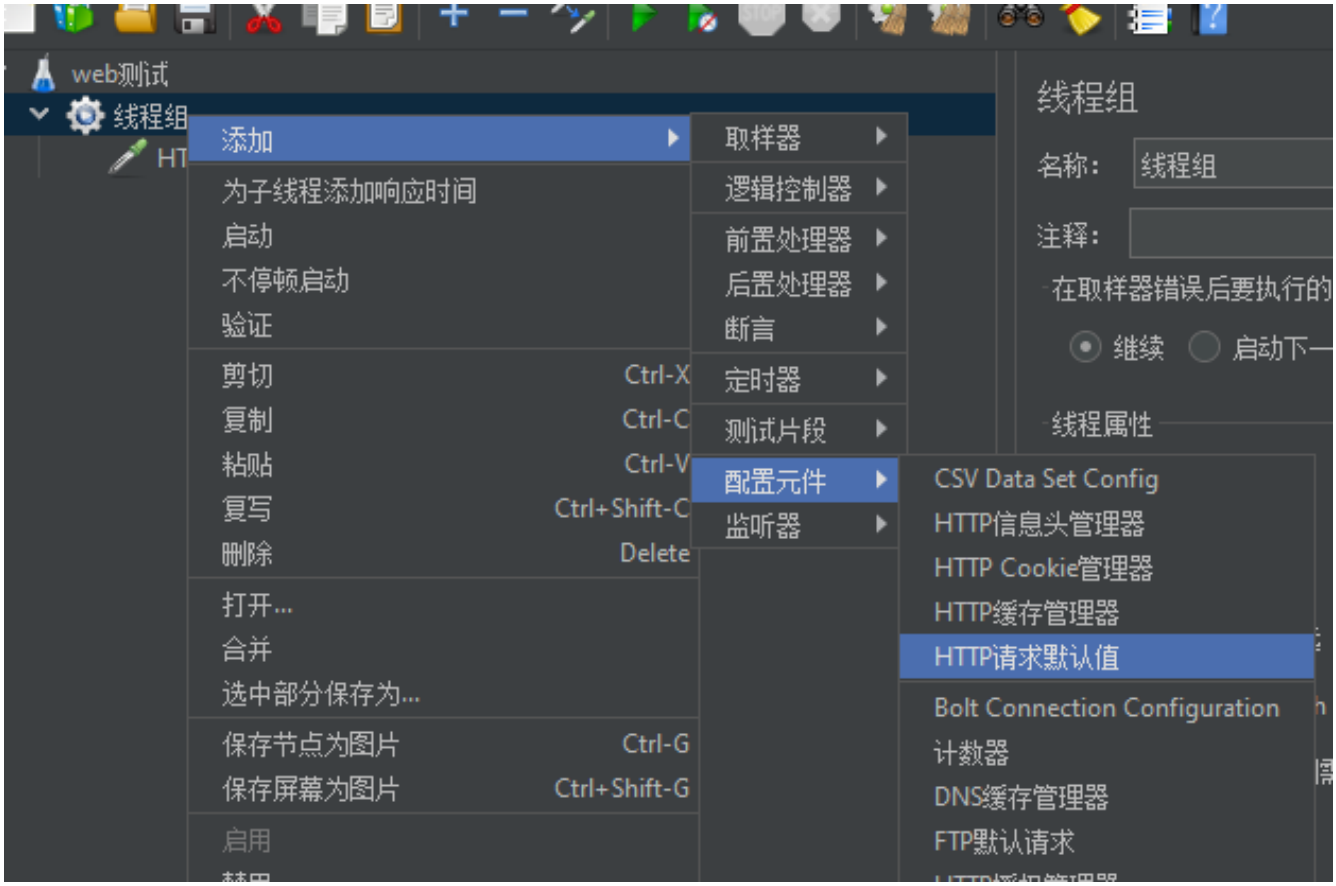
解压，运行jmeter.cmd（windows）

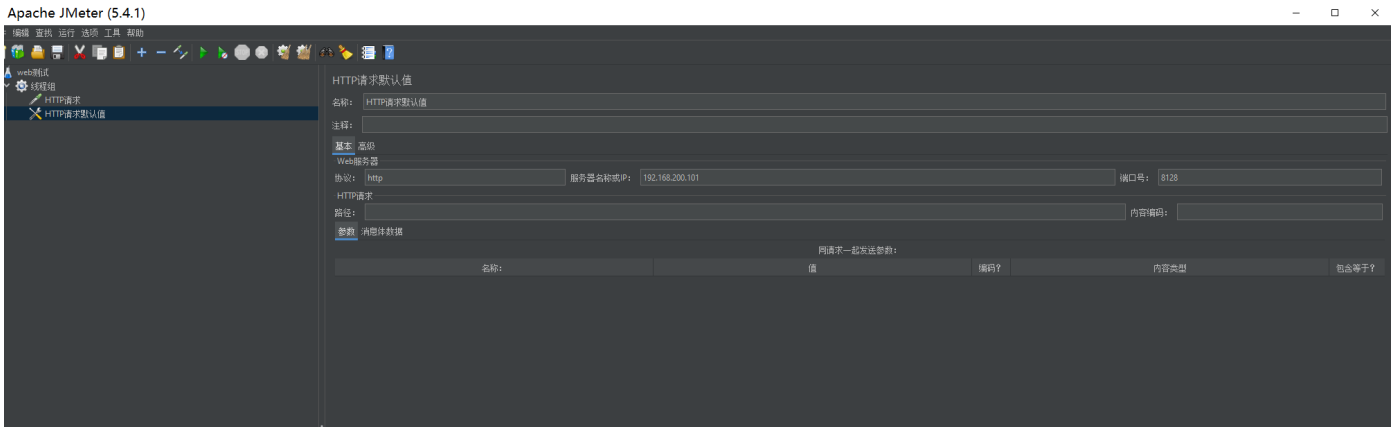


添加线程组：

线程数 设置为：500

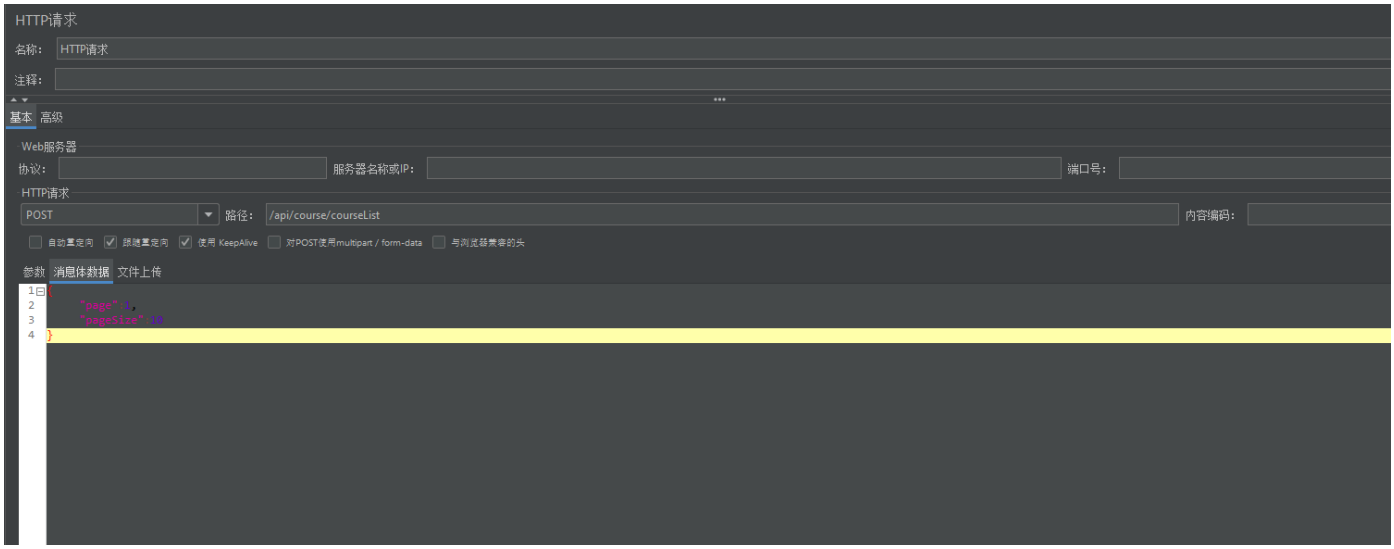
设置请求默认值，为所有请求添加一些公共配置


web测试
线程组
添加
为子线程添加响应时间
启动
不停顿启动
验证
剪切 Ctrl-X
复制 Ctrl-C
粘贴 Ctrl-V
复写 Ctrl+Shift-C
删除 Delete
打开...
合并
选中部分保存为...
保存节点为图片 Ctrl-G
保存屏幕为图片 Ctrl+Shift-G
启用
取样器
逻辑控制器
前置处理器
后置处理器
断言
定时器
测试片段
配置元件
监听器
CSV Data Set Config
HTTP信息头管理器
HTTP Cookie管理器
HTTP缓存管理器
HTTP请求默认值
Bolt Connection Configuration
计数器
DNS缓存管理器
FTP默认请求
线程组
名称: 线程组
注释:
在取样器错误后要执行的
继续
线程属性

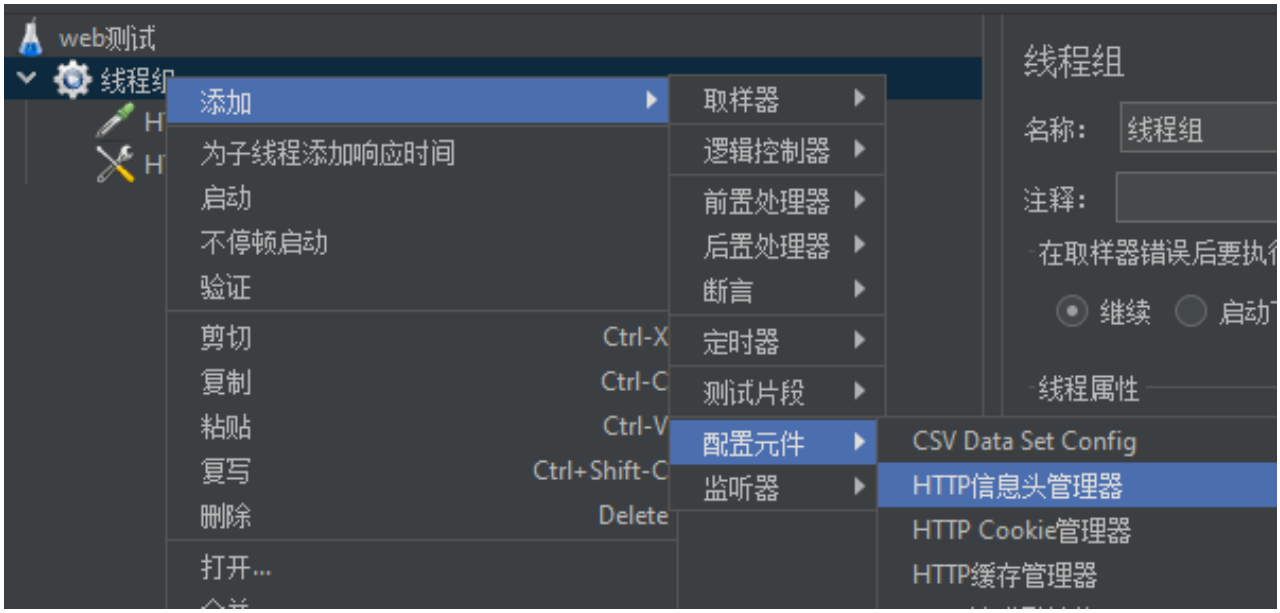

Apache JMeter (5.4.1)
HTTP请求默认值

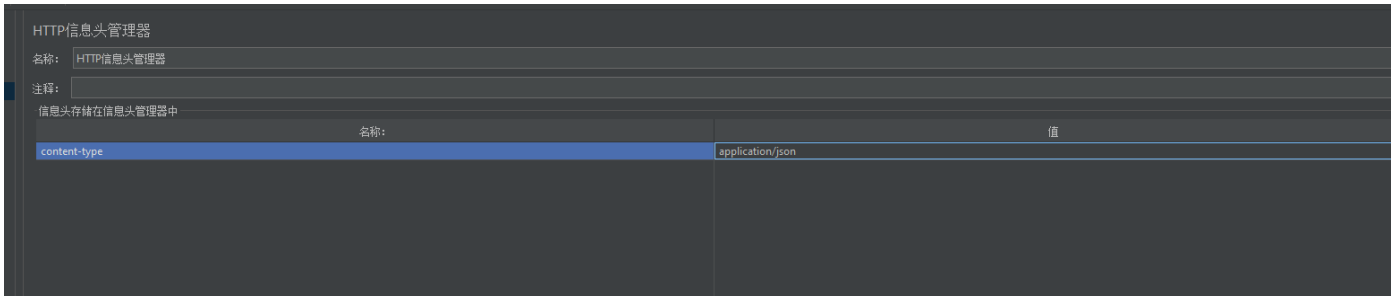
构造HTTP请求


件 编辑 查找 运行 选项 工具 帮助
web测试
线程组
添加
为子线程添加响应时间
启动
不停顿启动
取样器
逻辑控制器
前置处理器
测试活动
HTTP请求
Debug Sampler
JSR223 Sampler
线程组

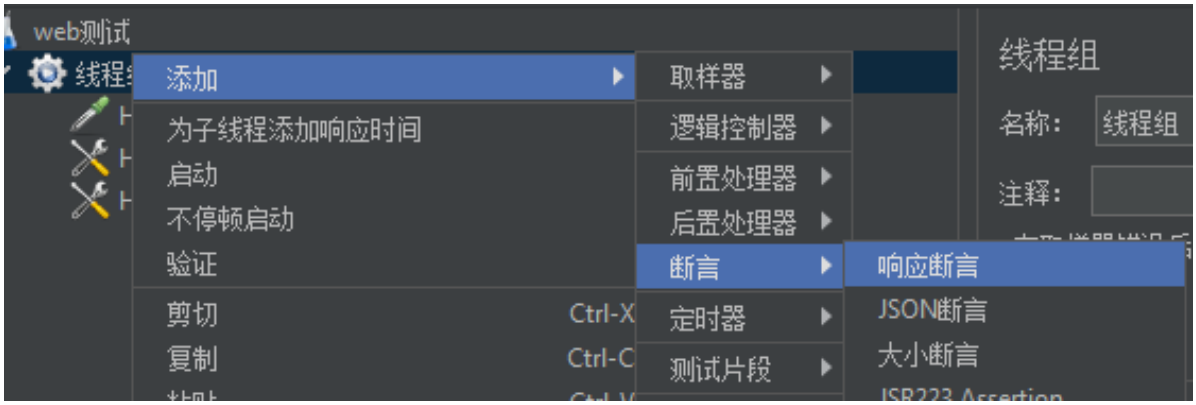


添加http请求头：

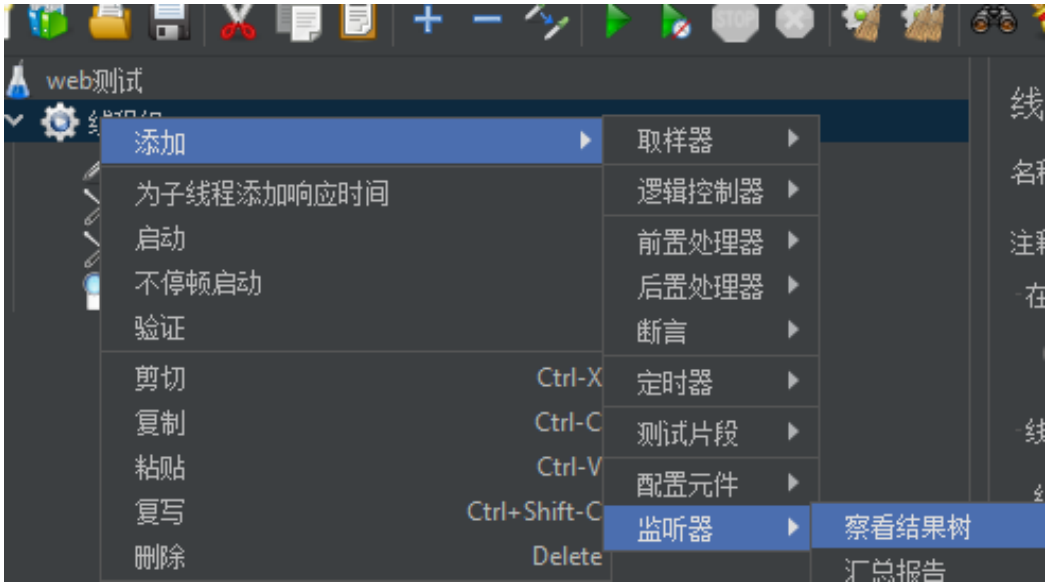




添加断言：

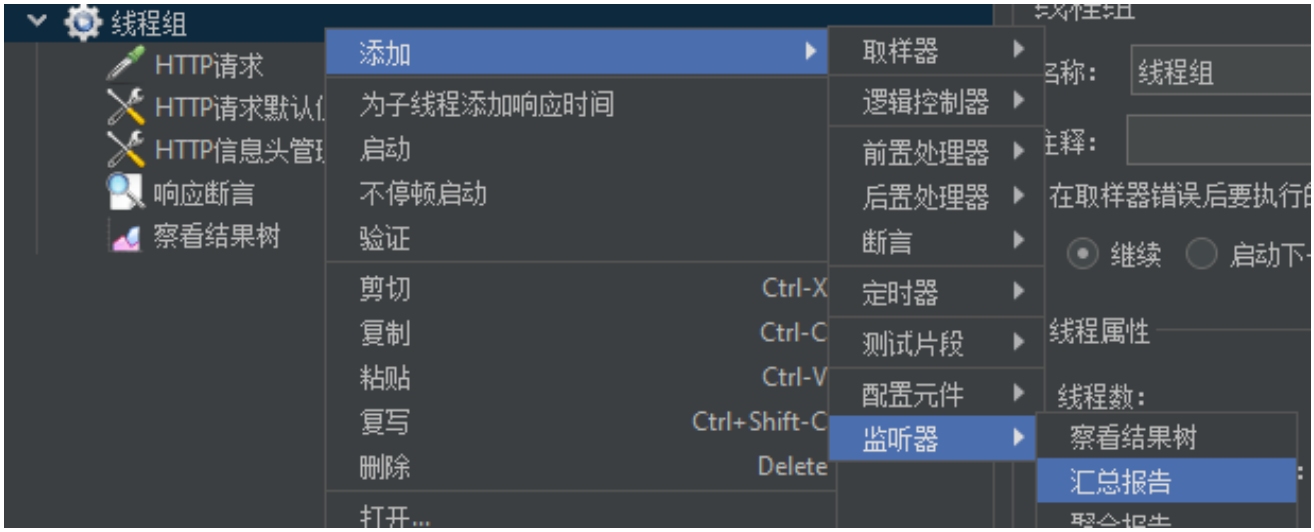


添加查看结果树：



点击运行，可以看一下请求是否正确。

添加加Summary Report：



点击运行可以查看结果。

2.3.2.1 执行测试计划

```
jmeter -n -t web测试.jmx -l web_test/result.txt -e -o web_test/report.out
```

2.3.3 分析

```
 [GC (Allocation Failure) 2021-12-02T21:40:40.605+0800: 0.833: [DefNew:
157248K->2109K(157248K), 2.4087729 secs] 359913K->221248K(506816K),
2.4089225 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
```

GC：如果前面没有Full修饰，代表这是一次Minor GC

Allocation Failure：本次引起GC的原因是因为在年轻代中没有足够的空间

DefNew: 串行收集器

157248K->2109K(157248K)：**单位是KB**

分别代表：GC前该内存区域(这里是年轻代)使用容量，GC后该内存区域使用容量，该内存区域总容量。

2.4087729 secs：该内存区域GC耗时，单位是秒

359913K->221248K(506816K)：三个参数分别为：堆区垃圾回收前的大小，堆区垃圾回收后的大小，堆区总大小。

2.4089225 secs：该内存区域GC耗时，单位是秒

Times: user=0.01 sys=0.00, real=0.02 secs：分别表示用户态耗时，内核态耗时和总耗时

分析：

年轻代此次GC减少了：157248-2109=155319K

Heap区总共减少了： 359913-221248=138665K

155319-138665=16654K, 代表共有这么16654K从年轻代移动到了老年代

```
Full GC (Metadata GC Threshold) 2021-12-02T21:40:45.578+0800: 5.806:
[Tenured: 24576K->30347K(349568K), 0.1040736 secs] 149544K-
>30347K(506816K), [Metaspace: 33668K->33668K(1081344K)], 0.1041350 secs]
[Times: user=0.11 sys=0.00, real=0.11 secs
```

`Metadata GC Threshold`:Metaspace大小达到了GC阈值

`Tenured`：老年代

`24576K->30347K(349568K)`:GC 前该区域已使用容量 -> GC 后该区域已使用容量 (该区域内存总容量)

`149544K->30347K(506816K)`:GC 前Java堆已使用容量 -> GC后Java堆已使用容量（Java堆总容量）

`Metaspace: 33668K->33668K(1081344K)`:元空间 使用内存gc前后的变化

> *通过日志可以看出，Metaspace区并没有真正释放空间，所以怀疑是Metaspace区不够用了。JDK8中，`XX:MaxMetaspaceSize`确实是没有上限的，最大容量与机器的内存有关；**但是`XX:MetaspaceSize`是有一个默认值的：21M**。*

解决：设置一个`XX:MetaspaceSize`的JVM启动参数：`-XX:MetaspaceSize=128M`

> *同时年轻代gc过于频繁，时间也较长，考虑设置的gc相关参数不合理，我们重新设置一个gc参数*

`-XX:+UseG1GC`： 设置G1垃圾回收器

`-XX:MetaspaceSize=128M`

`-Xms1024m`:最小堆内存

-Xmx1024m：最大堆内存

-Xmn384m:新生代大小

-XX:NewRatio:默认2表示新生代占年老代的1/2，占整个堆内存的1/3。

-XX:SurvivorRatio:默认8表示一个survivor区占用1/8的Eden内存，即1/10的新生代内存。

-XX:+PrintAdaptiveSizePolicy：自适应策略，用于G1调优

```
java -jar -Xmx1024m -Xms1024m -Xmn384m -XX:+UseG1GC -
XX:MetaspaceSize=128M -XX:+HeapDumpOnOutOfMemoryError -
XX:+PrintAdaptiveSizePolicy -XX:HeapDumpPath=sso.dump  -
XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:sso-api.gc
```

```
[GC pause (G1 Evacuation Pause) (young), 0.0843619 secs]
   [Parallel Time: 83.7 ms, GC Workers: 1]
      [GC Worker Start (ms):  258031.6]
      [Ext Root Scanning (ms):  4.4]
      [Update RS (ms):  15.9]
         [Processed Buffers:  101]
      [Scan RS (ms):  0.6]
      [Code Root Scanning (ms):  0.0]
      [Object Copy (ms):  62.6]
      [Termination (ms):  0.0]
         [Termination Attempts:  1]
      [GC Worker Other (ms):  0.0]
      [GC Worker Total (ms):  83.6]
      [GC Worker End (ms):  258115.2]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.1 ms]
   [Other: 0.5 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.1 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.0 ms]
      [Humongous Register: 0.0 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 0.1 ms]
   [Eden: 355.0M(355.0M)->0.0B(354.0M) Survivors: 29.0M->30.0M Heap:
752.5M(1024.0M)->405.0M(1024.0M)]
```

```
   [Times: user=0.05 sys=0.03, real=0.08 secs]
```

young:年轻代垃圾回收情况

[Parallel Time: 83.7 ms, GC Workers: 1]:标记着并行阶段的汇总信息。总共花费时间以及GC的工作线程数。

[GC Worker Start (ms): 258031.6]:开始时间

Ext Root Scanning (ms): 4.4: 外部根区扫描。外部根是堆外区。JNI引用，JVM系统目录，Classloaders等

Update RS (ms): 15.9:RSet的处理, UpdateRS:更新RSet的时间信息。-**XX:MaxGCPauseMillis**(默认200ms)参数是限制G1的暂停时间，一般RSet更新的时间小于10%的目标暂停时间是比较可取的。

Code Root Scanning (ms): 0.0:代码根的扫描

Object Copy (ms): 62.6: 该任务主要是对CSet中存活对象进行转移（复制）。对象拷贝的时间一般占用暂停时间的主要部分。如果拷贝时间和”预测暂停时间“有相差很大，也可以调整年轻代尺寸大小。

Termination (ms): 0.0: 终止工作线程。Work线程在工作终止前会检查其他工作线程的任务，如果其他work线程有没完成的任务，会抢活。如果终止时间较长，可能是某个work线程在某项任务执行时间过长。

GC Worker Other (ms): 0.0:花在GC之外的工作线程的时间，比如因为JVM的某个活动，导致GC线程被停掉。这部分消耗的时间不是真正花在GC上，只是作为log的一部分记录。

GC Worker Total (ms): 83.6:并行阶段的GC汇总，包含了GC以及GC Worker Other的总时间

GC 串行活动:

```
   [Code Root Fixup: 0.0 ms]
     [Code Root Purge: 0.0 ms]
     [Clear CT: 0.1 ms]
```

串行的GC活动。包括代码根的更新和扫描。Clear的时候还要清理RSet相应去除的Card Table信息。G1 GC在扫描Card信息时会有一个标记记录，防止重复扫描同一个Card。

GC Other活动:

剩余的部分就是其他GC活动了。主要包含:选择CSet、引用处理和排队、卡片重新脏化、回收空闲巨型分区以及在收集之后释放CSet

```
   [Other: 0.5 ms]
        [Choose CSet: 0.0 ms]
        [Ref Proc: 0.1 ms]
        [Ref Enq: 0.0 ms]
        [Redirty Cards: 0.0 ms]
        [Humongous Register: 0.0 ms]
        [Humongous Reclaim: 0.0 ms]
        [Free CSet: 0.1 ms]
```

垃圾收集结果统计: [Eden: 355.0M(355.0M)->0.0B(354.0M) Survivors: 29.0M->30.0M Heap: 752.5M(1024.0M)->405.0M(1024.0M)]

`Eden: 355.0M(355.0M)->0.0B(354.0M)`: Eden分区GC前355M,GC后是0,括号里面的分别是GC前后Eden分区的总大小。可以看到在一次GC后，Eden的空间做了调整。G1 GC的暂停时间是可预测的，所以YoungGC之后，会根据pause time的目标重新计算需要的Eden分区数，进行动态调整。

`Survivors: 29.0M->30.0M`: Survivors空间的变化，空间增长了，说明有存活对象E区晋升到S区。

`Heap: 752.5M(1024.0M)->405.0M(1024.0M)`：整个堆区的GC前后空间数据，G1 GC会动态调整堆区，但这次回收中没有改变堆区的容量。

年轻代调优:

因为G1 GC是启发式算法，会动态调整年轻代的空间大小。目标也就是为了达到接近预期的暂停时间。年轻代调优中比较重要的就是对暂停时间的处理。一般都是根据MaxGCPauseMillis以及年轻代占比G1NewSizePercent、G1MaxNewSizePercent，结合应用的特点和GC数据进行接近期望pause time的调整。

```
 26.139: [GC pause (G1 Evacuation Pause) (young) 26.139: [G1Ergonomics
(CSet Construction) start choosing CSet, _pending_cards: 3484, predicted
base time: 5.51 ms, remaining time: 194.49 ms, target pause time: 200.00
ms]
 26.139: [G1Ergonomics (CSet Construction) add young regions to CSet,
eden: 54 regions, survivors: 9 regions, predicted young region time:
5.98 ms]
 26.139: [G1Ergonomics (CSet Construction) finish choosing CSet, eden:
54 regions, survivors: 9 regions, old: 0 regions, predicted pause time:
11.49 ms, target pause time: 200.00 ms]
, 0.0163685 secs]
```

> *target也即目标是200ms,实际的pause time是16ms。远远小于目标暂停时间。并且再CSet中的分区数是“eden: 54 regions, survivors: 9 regions”，可以适当增加CSet中的年轻代分区,也可以适当缩短暂停时间，让实际值和期望值不断接近。*

2.4 推荐配置

JVM推荐配置原则：

> 1. *应用程序运行时，计算老年代存活对象的占用空间大小X。程序整个堆大小（Xmx和Xms）设置为X的3 ~ 4倍；永久代PermSize和MaxPermSize设置为X的1.2 ~ 1.5倍。年轻代Xmn的设置为X的1 ~ 1.5倍。老年代内存大小设置为X的2 ~ 3倍。*
> 2. *JDK官方建议年轻代占整个堆大小空间的3/8左右。*
> 3. *完成一次Full GC后，应该释放出70%的堆空间（30%的空间仍然占用）。*

3. 加密

> *部署上线的应用，密码在配置文件中明文，这样是极为不安全的，所以我们需要对密码进行加密处理*

我们使用`jasypt`来对application.properties配置文件中的mysql账号和密码进行加密

1. 引入依赖

```
<dependency>
        <groupId>com.github.ulisesbocchio</groupId>
        <artifactId>jasypt-spring-boot-starter</artifactId>
        <version>2.1.1</version>
</dependency>
```

2. 将字符串进行加密

```
package com.mszlu.xt.web.config;

import lombok.extern.slf4j.Slf4j;
import org.jasypt.util.text.BasicTextEncryptor;

@Slf4j
public class TestEncode {


    public static void main(String[] args) {
        BasicTextEncryptor textEncryptor = new
BasicTextEncryptor();
        //加密所需的salt(盐)
        textEncryptor.setPassword("mszlu");
        //要加密的数据（数据库的用户名或密码）
        String username = textEncryptor.encrypt("root");
        String password = textEncryptor.encrypt("root");
        log.info("username:{}",username);
        log.info("password:{}",password);
    }
}
```

3. 替换配置文件中数据库账号密码的部分

```
#数据库配置
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/xt?
useUnicode=true&characterEncoding=utf-8&serverTimezone=UTC
spring.datasource.username=ENC(PmgAc8SnQp2AWWI7l2I78w==)
spring.datasource.password=ENC(wkYb0BqIvK/hArr5it/lDg==)
```

4. 运行jar的时候，外部指定加密盐

```
java -jar -Djasypt.encryptor.password=mszlu xx.jar
```

> *上线前准备已经做完，可以开始上线了*