首先通过下面两条sql及打印的执行sql，清楚明了的看一下它们的区别：

```
<select id="selectUserInfo" parameterType="java.util.Map"
resultType="java.util.Map">
    select
        *
    from
        user
    where
        userId=${id} password=#{pwd}
</select>
```

假设入参传入的是1，打印执行sql如下：

```
Preparing:select * from user where id=1 and password=111111
```

```
<select id="selectUserInfo" parameterType="java.util.Map"
resultType="java.util.Map">
    select
        *
    from
        user
    where
        userId=#{id} and password=#{pwd}
</select>
```

同样入参传入1，打印的执行sql如下：

```
Preparing:select * from user where id=?  and password=?
Parameters:1(String),111111(String)
```

MyBatis启用了预编译功能，在SQL执行前，会先将上面的SQL发送给数据库进行编译；执行时，如果入参为#{}格式的，将入参替换编译好的sql中的占位符“?”；如果入参格式为${}，则直接使用编译好的SQL就可以了。因为SQL注入只能对编译过程起作用，所以使用#{}入参的方式可以很好地避免了SQL注入的问题。

mybatis预编译底层实现原理

MyBatis是如何做到SQL预编译的呢？其实在框架底层，是JDBC中的PreparedStatement类在起作用，PreparedStatement是我们很熟悉的Statement的子类，它的对象包含了编译好的SQL语句。这种“准备好”的方式不仅能提高

安全性，而且在多次执行同一个SQL时，能够提高效率。原因是SQL已编译好，再次执行时无需再编译。

总结

#{}：相当于JDBC中的PreparedStatement

${}：是输出变量的值

简单说，#{}是经过预编译的，是安全的；${}是未经过预编译的，仅仅是取变量的值，是非安全的，存在SQL注入。

番外（sql注入）

还是以上面的两条sql为例，入参id的值传入“1 or userId=2”,入参pwd的值传入“111111”。以#{}格式传入入参后的执行sql：

select * from user where userId=”1 or userId=2” and password = “111111”;

以${}格式传入入参后的执行sql：

select * from user where userId=1 or userId=2 and password = 111111;

很显然，**以${}格式传入入参后的执行sql打乱了我们的预期sql格式及查询条件，从而实现sql注入。所以，除了order by 等需要传入数据库字段等的入参使用${},其他的尽量使用#{}。**